A Method for Understanding Experimental Computer Programs in Artificial Intelligence Research

Amaia Bernaras Iturrioz

Ph.D.

University of Edinburgh

1993

Zuri ama, nere exenplu eta gidari izateagatik.

I dedicate this thesis to my mother,
for her example and guidance.

Abstract

Tradition has it that work in Software Engineering has little or nothing to offer Artificial Intelligence (AI) research. The argument is that building computer programs in AI is very different from building computer programs for application. Although I accept this, I believe that some Software Engineering concepts and constructs can be used to improve the research done in AI, in particular in the Symbolic Paradigm. This paradigm is based upon Newell and Simon's Physical Symbol System Hypothesis, which states that a physical symbol system has the necessary and sufficient means for general intelligent action. Experiments in this paradigm involve building experimental computer programs to investigate whether a physical symbol system can be further organised to exhibit particular kinds of intelligent action. As part of the experimental procedure, it is necessary to identify and to understand the components and structure of a program that cause its observed behaviour. This involves a static analysis of the program (in contrast to a dynamic analysis which is concerned with testing the scope of its behaviour). To understand a program's structure directly from an inspection of the actual code can be very difficult. It is likely to be easier to understand a form of a program in which the significant features of the program are described in a way abstracted from implementational details.

This thesis is concerned with the use of Software Engineering abstraction constructs to help in the process of understanding computer programs that are built as part of experiments in the Symbolic Paradigm. It is also concerned with developing and testing a method to analyse these programs in an organised and structured way. In a series of three experiments, the use of abstraction constructs to help the process of transforming a program to a more abstract form, and how to do this in a structured way, was incrementally investigated. This involved first exploring the use of abstraction constructs to achieve higher degrees of abstraction in a small example; the next step was to use the understanding of their use and of how to transform a program in a bigger example, from which a more clearly defined role of the abstraction constructs, and an initial scheme for transforming a program, was achieved; the last step involved investigating a complete form of an analysis procedure to analyse experimental programs built by incremental prototyping, and that is supported by the use of abstraction constructs. The result is the Prototype Analysis Method (PAM): a static analysis method to help in the understanding of incrementally built experimental computer programs in

AI. An essential part of this method is a transformation process that is supported by the use of Software Engineering abstraction constructs, and of test sets from dynamic analysis for validation.

This research clearly demonstrates the succesful application of Software Engineering abstraction constructs in an important aspect of AI research. Results from this research also point to further interesting issues such as the relation between Knowledge Level descriptions and abstract Symbol Level descriptions.

The composition of this thesis and the research reported in it are entirely my own work, except where otherwise stated.

Amaia Bernaras Iturrioz

Table of Contents

List of Figures

Chapter 1

Introduction

Tradition has it that work in Software Engineering has little or nothing to offer Artificial Intelligence (AI) research, the argument being that program building in AI is very different from application program building. I accept that they are different, but I also believe that some Software Engineering concepts and constructs can be used to improve research in AI, in particular in the *Symbolic Paradigm.* The Symbolic Paradigm in AI (also called Symbolic Functionalism) is based upon Newell and Simon's *Physical Symbol System Hypothesis* (PSSH), [Newell & Simon 76], which states that a physical symbol system has the necessary and sufficient means for general intelligent action. Here, general intelligent action means the scope of intelligence displayed by humans. The investigation of the sufficiency part of this hypothesis (i.e., that any physical symbol system of sufficient size can be organised further to exhibit general intelligent action) motivates much of the research done in this paradigm. Typically, experiments in this paradigm involve building computer programs. However, attempting to achieve general intelligent action (of the scope seen in humans) by further organising a physical symbol system in any one program is not very realistic. Instead, experiments typically involve building computer programs to investigate how a physical symbol system can be further organised to exhibit particular kinds of intelligent action (chess playing, or expert diagnosis, for example).

Just building a computer program that displays some kind of intelligent behaviour (i.e., just organising further a physical symbol system) is, however, not enough to understand what is required to achieve the kind of intelligent behaviour being investigated. What is important about any one experiment is to identify clearly and accurately what is the further organisation (of a symbol system) that

is needed to achieve a particular kind of intelligent behaviour: so, just building computer programs is not enough. It is also necessary to understand how the physical symbol system has been further organised in the program, that is, what is the program's causal structure: the structures, and the processes operating on them that are responsible for producing the program's behaviour. Understanding a program's causal structure is important in achieving the kind of understanding we seek in science and to allow replication or development of the program's essential ideas. It is also important in the broader context of understanding computer programs in general, and although I have investigated this issue in the context of the Symbolic Paradigm in AI, results from my investigations can be useful for program understanding in general.

Typically, an experimental program turns out differently from what was intended at the beginning of the building process. The program changes as the researcher understands better the task involved and how to achieve it. An experiment is of no use if it is assessed purely in terms of the original intentions, rather than actual results. Thus, for an experiment to be useful, it is important to understand the program actually built rather than the program initially intended by a researcher. Understanding the resulting program is not an easy task, and one that is not usually performed by researchers in AI.

In this thesis, I present a framework for performing experiments, in Symbol Processing AI, and which incorporates a level in which an abstract description of the causal structure of experimental programs is presented—the Symbol Level. This framework is based on three levels of understanding: Knowledge Level, Symbol Level, and System Engineering Level. The Knowledge Level was first presented by Newell [Newell 81], and the Symbol Level and System Engineering Level are developments of the Symbol Level as described by Newell [Newell 81], and they are related to Brachman and Levesque's levels for Knowledge Based Management Systems [Brachman & Levesque 86], although there are significant differences as we will see in chapter 2. As part of this framework, programs are not just built but they are also analysed afterwards, so that they are understood better.

To achieve this understanding it is necessary to identify and understand the components and structure of a program that cause its observed behaviour. This involves a static analysis of it, rather than a dynamic analysis, which is necessary to test the program's behaviour. To understand a program's structure directly

from inspecting the actual code can be very difficult, as the experience from building the Edinburgh Designer System showed (see section 1.1). A program is usually easier to understand if an abstract description of it is available, that is, a form of the program where the significant features of the program are described in a way abstracted from implementational details (which are irrelevant to understanding its causal structure). In this more abstract form, the program is likely to be easier to understand on inspection.

An abstract description of this kind can be obtained by transforming a program in the right way, and Software Engineering *abstraction constructs* might be useful in this process of program transformation. Software Engineering research is concerned with facilitating aspects of software production, and in particular program building. This involves the development of programming languages and environments that make the program building task easier. Modern programming languages achieve this by incorporating abstraction constructs that provide more abstract levels of programming. Abstraction constructs are developed to help the process of going from a high level (more abstract) description of a program, to actual program code, and their support is based on the abstraction they provide. Although the purpose of these abstraction constructs is to facilitate program building, the abstraction they provide may also make them useful in program analysis, that is, in going from the program to a more abstract form of it.

This thesis is concerned with investigating the use of some of these abstraction constructs to help in obtaining a better understanding of AI research programs. I have investigated their use (as they are implemented in the Napier88 persistent programming language [Morrison *et al* 89]) to support the transformation of a program to a form that is easier to understand on inspection. This transformation process forms an essential part of a static analysis procedure (the Prototype Analysis Method) which was developed and investigated to help in understanding incrementally built experimental AI programs.

In the rest of the chapter I will present the motivation of this work, and the thesis I set out to investigate. I will also present the aims and objectives of this investigation, and the major achievements. A summary of the rest of the thesis will follow.

1.1 Motivation and Thesis

Before starting my Ph.D., I spent a year as a visiting researcher in the Design Research Group of the Artificial Intelligence Department, at the University of Edinburgh. During that year, I became familiar with the Edinburgh Designer System, also known as EDS. The Edinburgh Designer System is an experimental design support system, that is, a system aimed at supporting a human designer in performing a design task. At the time of my arrival, the group was engaged in building the seventh and last prototype of the system for the Alvey large scale demonstrator 'Design to Product' [Smithers 87]. From conversations with members of the group it became clear that understanding the system was a difficult task. Even for the builders it was difficult to understand how it was that the system behaved in the way it did. This difficulty was increased by the many incremental changes incorporated into the system during the various prototypes. New prototypes usually contained a mixture of old prototypes (or parts of them) and new features being investigated. It was during that year that I became interested in the problem of understanding AI research programs, particularly those built by incremental prototyping.

Also during that year, I came across an interesting paper written by Ron Morrison, Professor in the Department of Computational Science, at the University of St. Andrews, and his colleagues. In this paper, [Morrison *et al* 88], it is argued that the Software Engineering and AI communities have underestimated each other's work for a long time, in particular the AI community, who still consider Software Engineering developments as not suitable for the adaptive, and complex data used in AI systems. It is argued that, although this could be true of more traditional Software Engineering languages, modern programming languages are suitable for AI because they incorporate abstraction constructs and mechanisms that are also appropriate for AI systems. I took on their challenge that the AI community should take modern developments in Software Engineering more seriously, and decided to investigate how they might usefully be used in the analysis of experimental AI programs.

Thus, motivated by the need to understand better AI research computer programs, and wanting to see whether Software Engineering abstraction constructs can be of use in doing AI, I set out to investigate the following thesis:

The use of abstraction constructs developed in the area of programming language research in Software Engineering can help to understand better experimental computer programs in AI research.

1.2 Aims, Objectives, and Achievements

The aim of my work was to investigate the use of abstraction constructs in the understanding of computer programs built as part of AI experiments performed to recreate some form of intelligent behaviour. To achieve this aim, two objectives were identified:

- To investigate the application of abstraction constructs found in modern Software Engineering programming languages in the analysis of AI-like programs. I call the programs used for my experiments 'AI-like programs' because they are examples of the kinds of programs built in AI research but they are simpler than real experimental programs. The abstraction constructs investigated were persistence (a property of data that determines how long data exists for) strong typing (a kind of typing that ensures that all expressions in a program are type consistent) abstract data types (which provides encapsulated representation of a complex data object and its operations) and polymorphism (which offers greater levels of data abstraction).

- To develop and investigate an organised way of applying the abstractions constructs and of performing the analysis; a method that leads to a better understanding of the programs analysed.

The major achievements of this work are:

- A successful application of some of the abstraction constructs investigated: *persistence* for providing information hiding and helping in managing the complexity of the analysis, *strong typing* for providing increased precision during the transformation of a program, and *structural equivalence* for helping in the renaming process during the analysis. On the other hand, *abstract data types* and *polymorphism* were not found to be helpful for the analysis: polymorphism does not provide the right kind of abstraction, and abstract data types adds complexity to the process.

- The development of a method for the analysis of AI research programs: the Prototype Analysis Method (PAM). PAM helps to analyse a program in an organised way, and it is supported by the use of abstraction constructs. It includes a process of program transformation that helps to transform a program to a more abstract form. This form is easier to understand by inspection at the domain level where interpretations are placed on data structures and operations based on the problem domain or problem solving method used. PAM also incorporates features for the analysis of incrementally built programs so that results of the analysis of earlier prototypes can be used in the analysis of later ones.

- A framework for performing AI experiments to replicate some behaviour. This framework identifies three levels which involve different aspects of an experiment. The Knowledge Level (where the knowledge required to display the intelligent behaviour investigated is described) the System Engineering Level (where a program is built to realise the behaviour described at the Knowledge Level) and the Symbol Level (where the representational role of the data structures and their associated operations in the program is described).

1.3 The Exciting Things to Come

Chapter two presents the background and context of this thesis. This includes the scientific and experimental nature of AI, and the role of experiments as part of the Symbolic Paradigm in AI. It also presents the framework, developed in this thesis, for performing experiments. Chapter three elaborates on the problem addressed in this thesis—the need to understand better AI research programs—and the approach taken to tackling it—the static analysis of these programs. Chapter four discusses some aspects that are important for the analysis and Software Engineering mechanisms that help to achieve them. It reviews different abstraction constructs that implement those mechanisms and shows the advantages of the ones chosen for the investigations. Chapter five presents the three experiments performed to test the use of abstraction constructs to help in the analysis of AI research programs. These experiments also involved investigating a way of analysing a program (including the application of the abstraction constructs). In the first experiment, the use of abstraction constructs

was explored to transform a program implementing a well specified problem. In the second experiment, the use of abstraction constructs was further explored to transform a more AI-like program (i.e., a program displaying what can be considered intelligent behaviour, and built without clear specifications). This exploration was performed in a more guided way, and an initial scheme to apply the abstraction constructs and to transform a program emerged. Finally, in the third experiment the use and role of abstraction constructs was further tested as part of a more complete form of the method developed to analyse AI programs. The program in this experiment was built by incremental prototyping, and its analysis was used to investigate the analysis of programs built by incremental prototyping. The description of each experiment includes its objectives, the work done to achieve them, results, assessment, observations, discussion, and outcome of the experiment. The method developed and tested in these experiments, PAM, is presented in chapter six. Chapter seven presents a discussion of the abstraction constructs and the method investigated. Chapter eight relates the results from the experiments to other aspects of the framework for performing experiments presented in chapter three. Finally chapter nine presents conclusions and future work.

Chapter 2

Artificial Intelligence as an Experimental Science

In this chapter I will present the background and context of the thesis. This involves presenting AI as an experimental science concerned with studying intelligent behaviour in the artificial, and discussing this view with respect to other sciences and the different kinds of AI, i.e., different paradigms in AI. I will then introduce the role of experiments in the Symbolic Paradigm. I will discuss relevant aspects of performing experiments including the role of programs and the need to analyse them to provide an appropriate description of the system built. I will propose a framework of three levels in which to understand experiments—The Knowledge Level (first proposed by Newell [Newell 81]) the Symbol Level, and the System Engineering Level. Related levels (such as those of Brachman and Levesque [Brachman & Levesque 86]) and frameworks will also be discussed. Finally I will discuss current AI practice in terms of these three levels.

2.1 A Description of AI

The proper nature and foundation of AI continues to be the subject of much debate, and rightly so for AI is still a relatively young discipline. For example, [Partridge & Wilks 90] presents a collection of papers from a workshop on the foundations of AI held in Las Cruces, New Mexico in 1986, together with some earlier material to be found in the AI literature. For another example see [Graubard 88] which presents a collection of invited papers first published in a *Daedalus* volume by the American Academy of Arts and Sciences. Many

different descriptions of AI are given in these and other papers and books in the AI literature. The one I prefer is given by Smithers in [Smithers 91]:

> *Artificial Intelligence is the science concerned with understanding intelligent behaviour by attempting to create it in the artificial.*

Considering AI as a science that attempts to understand intelligent behaviour by creating and investigating it in the artificial presumes a number of things. It presumes that intelligent behaviour is a phenomenon that can be investigated scientifically (AI is a science concerned with understanding intelligent behaviour) and that artificial systems can be made to behave intelligently (intelligent behaviour is investigated by creating it in the artificial). This description also implies that AI is an experimental science, rather than an observational one. In this section I elaborate on this description of AI by considering these aspects and associated assumptions in more detail.

2.1.1 AI as a Science

In considering AI as a science, I first explain my understanding of science. I distinguish natural sciences from sciences of the artificial, and also experimental from observational ones. Finally, I characterise AI as an experimental science of the artificial.

What is a Science?

The overall aim of a science is the development of theories about the phenomena under study. A scientific theory can be characterised as some statement of a law about some aspect of the phenomenon under investigation, and which is expressible in a way independent of, and without reference to, particular examples or instances of the phenomenon. When possible, laws are expressed in science by providing formal (abstract) descriptions of them (e.g., Newton's laws of motion). Thus in science we try to understand things by seeking to develop formal (abstract) descriptions of the laws which govern the form of the phenomena under investigation.

Although the above description can be applied to science in general, useful distinctions can be made to help the characterisation of particular sciences.

Aspects such as the kinds of phenomena a science investigates and the way the investigation is carried out are important to understand particular sciences.

Experimental or Observational: A distinction can be made between sciences that are mainly experimental and those that are mainly observational. In an experimental science, like physics or chemistry, phenomena are normally investigated by performing experiments. Experiments usually involve the manipulation of elements that are believed to contribute to the phenomenon investigated. For example, experiments to understand the molecular composition of water in chemistry can involve decomposing water into oxygen and hydrogen (by means of electricity). (See section 2.3 for more on experiments in science).

In an observational science, like astronomy, phenomena are investigated mainly by observation, rather than manipulation. Astronomers base their understanding of stars and their behaviour (as part of the Universe) on observation. They are not currently able to manipulate objects outside our solar system in any way.

Natural or Artificial: Sciences can also be divided into what are called natural sciences and sciences of the artificial. Natural sciences like physics, biology, chemistry, and astronomy seek to understand phenomena as they occur in the natural, in nature.

Sciences of the artificial, as described by Simon in [Simon 81], are concerned with the understanding of phenomena in the artificial, that is, phenomena created by us, humans. An example of a science of the artificial is computer science which studies computation by creating it in the artificial, i.e., with computers, [Smithers 91].

AI, a Science of the Artificial

Both in the description of AI given above and in [Simon 81], AI is understood to be a science of the artificial. Although it studies intelligent behaviour, which is also a naturally occurring phenomenon, it investigates it by creating artificial systems that display some kind of intelligent behaviour. This kind of investigation is different to that of other sciences studying naturally occurring examples of intelligent behaviour, such as psychology and cognitive science. By

studying intelligent behaviour in artificial systems, AI leads to a different, and complementary, type of understanding to that obtained by studying natural forms of intelligent behaviour.

AI as an Experimental Science

Herbert Simon was among the first researchers in AI to consider AI as an experimental science [Simon 81]. Its experimental nature is also stressed by Buchanan, [Buchanan 88], who points to Newell and Simon's General Problem Solver (GPS), [Newell & Simon 72], as an example of an early good AI research experiment. Buchanan also reports results from his own experiments in AI research. As an experimental science of the artificial, AI seeks to develop and investigate formal descriptions of the laws governing the intelligent behaviour created by artificial systems.

2.1.2 The Investigation of Intelligent Behaviour

Saying that AI is a science concerned with the understanding of intelligent behaviour assumes that intelligent behaviour is a phenomenon that can be investigated scientifically. This assumption seems reasonable if we look at the long standing work of other sciences. Investigations of naturally occurring examples of intelligent behaviour in other disciplines, i.e., cognitive psychology (human) and cognitive ethology (animal) have led to the successful development of useful scientific theories.

Intelligent Behaviour in the Artificial

Whether intelligent behaviour can be created in the artificial is an empirical question. Evidence from current work in AI supports an affirmative answer to this question, at least to some extent. This evidence comes from work on building systems that display some kind of intelligent behaviour—chess playing computer programs like HITECH, [Berliner & Ebeling 89], and ping-pong playing robots, [Andersson 88], amongst others. I expand on the kinds of systems built in AI in the next section.

2.2 Paradigms in AI

I have described AI as an experimental science that seeks to understand intelligent behaviour by creating it in the artificial, and I have discussed various aspects and assumptions behind this description. In this section, I discuss the different kinds of work currently done in AI, and also what I think AI is not. This will show that most research in AI can be understood in terms of this description, in particular, work in the Symbolic Paradigm (the context of this thesis).

2.2.1 Epistemological and Heuristic

In the paper 'Some philosophical problems from the standpoint of Artificial Intelligence', [McCarthy & Hayes 69], McCarthy and Hayes address some philosophical questions of what they call the 'epistemological' and 'heuristic' parts of AI, parts that are derived from their own understanding of the nature of intelligence.

Epistemological Programme

The epistemological part is concerned with models of the world. That is, it is concerned with understanding what aspects of the world to represent, and how to represent them, so that the solution of problems follows from the facts expressed [McCarthy & Hayes 69]. The work of McCarthy, Hayes, and others on this epistemological part is what is known as the epistemological programme,[1] and it is almost entirely theoretical.

The description of AI given in section 2.1 hardly applies to this work. Epistemologicists are not directly concerned with the investigation of intelligent behaviour, although they address important aspects concerning it (what to represent and how). Creating intelligent behaviour in the artificial is not one of their concerns either, although their work is likely to be useful when building systems that display some kind of intelligent behaviour [Cohen 91].

---

[1] They are also known as Logicists because logic is predominantly the formalism used for representation.

Thus the aim of the epistemological programme is to understand what knowledge can be formally represented and how it needs to be represented to solve a problem, independently of the actual system that uses that information to solve the problem. For a review of some epistemological problems see [McCarthy 85], and for some doubts about the progress being made on these questions see [McDermott 87].

Heuristic Part

According to McCarthy and Hayes, the 'heuristic' part is concerned with building the mechanism or machine that, on the basis of the knowledge it has, decides what to do to solve a problem. Other people, [Schank 90, Charniak & McDermott 85, Rich 83], take this other part of AI as the starting point, and consider that the aim of AI is to build an intelligently behaving machine, that is, a machine that can be seen (usually by us, humans) to show some kind of intelligent behaviour.

The form such a machine can take varies. I use this variation to introduce what are usually understood as different paradigms in AI—Symbolic Functionalism, Connectionism, and Robotic Functionalism.[2] Using the form that the machine takes in these three paradigms should not be understood as the defining difference between the paradigms, only as a useful way to talk about them.

As part of the Symbolic Paradigm, I discuss work on computational AI (also called basic AI). This discussion shows that not all work in Symbolic Functionalism is intended to be scientific in the way I have described in section 2.1. I also discuss two other views of AI—AI as computer science and AI as engineering.

2.2.2 Symbolic Functionalism

Symbolic Functionalism, based upon Newell and Simon's Physical Symbol System Hypothesis (PSSH), [Newell & Simon 76], presumes that symbol processing is the essence of intelligence, and that by processing the right symbols in the right way intelligent behaviour is produced. The PSSH has been, at least implicitly, adopted

---

[2]This third kind of functionalism in AI is rather recent, and the term was coined by Harnad [Harnad 89].

by most researchers working on symbol processing AI research since its earliest days.[3] The hypothesis states that a physical symbol system has the necessary and sufficient means for general intelligent action. In general terms, a physical symbol system consists of a set of entities, called symbols, and a set of processes that operate on the symbols (for more on physical symbol systems see [Newell 80] and [Harnad 90]).

According to Newell and Simon the term *necessary* means that any system that exhibits general intelligence will prove upon analysis to be a physical symbol system. *Sufficient* means that any physical symbol system of sufficient size can be organised further to exhibit general intelligence. The expression *general intelligent action* denotes the same scope of intelligence as seen in human action: that in real situations behaviour appropriate to the ends of the system and adaptive to the demands of the environment can occur, within some physical limits. Newell and Simon view AI as addressing itself to the sufficiency of physical symbol systems for engendering intelligent behaviour.[4] Usually symbol processing machines are implemented using programs that run on Von Neumann-type serial computers, [Newell & Simon 76], [Barr & Feigenbaum 81], [Rich 83], [Winston 77], and [Jackson 85].

Computational AI

Some practitioners in the Symbolic Paradigm understand AI not as a science but as a computational discipline. The aim of research in AI is then to explore computational techniques which have the potential for simulating intelligent behaviour [Bundy 90]. In this view AI is not a science like physics; it is more a discipline like applied mathematics. This is an interesting position, but one I disagree with. I agree that the study of computational techniques is an important part of the scientific investigation in AI research. However, as I have explained in section 2.1 I don't believe that it is all there is to it.

---

[3]Newell and Simon's work on the General Problem Solver (GPS) is an example of early experimentation to test the (at the time unstated) PSSH. When the hypothesis was stated, work on GPS was presented as evidence for the PSSH [Newell & Simon 76].

[4]They attribute to cognitive psychology the study of the *necessity* of having a physical symbol system whenever general intelligence is exhibited.

2.2.3 Other Paradigms

Connectionism The Connectionist Paradigm presumes that the essence of intelligent behaviour (or at least of cognition) is in the connections between large numbers of simple processing units, [Smolensky 88]. Connectionists are interested in building (or simulating) machines that contain a large interconnected collection of simple processors (e.g., binary threshold devices). Connectionist machines are inherently parallel, and they are best suited to investigation via parallel architectures (as opposed to the Von Neumann serial architecture) though they are often simulated on serial computers.

Robotic Functionalism Robotic Functionalism is concerned with understanding the relationship between agent, task, and environment, [Smithers 91]. To obtain this understanding robotic functionalists presume that they have to build real robots which do real things in real environments. Thus, their machines take the form of robots.

2.2.4 Building Systems in AI

In general, the description I have given of AI (section 2.1) can be taken as a characterisation of the work done in the 'heuristic' part of AI, and thus, in the paradigms I have described. Research in these areas is concerned with building, in the artificial, systems that display some kind of intelligent behaviour. Whether this research is practised as a science concerned with the study of intelligent behaviour is part of an ongoing debate in AI. In section 2.3 I discuss the scientific nature and practice of work done in the paradigm of interest to this thesis, Symbolic Functionalism.

2.2.5 What AI is Not

AI is sometimes understood to be a part of computer science. It is also sometimes understood as an engineering activity. Here I argue against these views.

AI is not Computer Science

Some researchers consider AI as a part of computer science (see [Charniak & McDermott 85] and [Barr & Feigenbaum 81] for example). Computer science studies computation and how it can be realized in computers. Viewed as a part of computer science, AI is characterised as studying the representation and use of symbolic information as opposed to using only numbers, and heuristic processes as opposed to strictly algorithmic processes [Buchanan 88]. This kind of characterization, and therefore the work that can be done following it, does not necessarily aim at the study of intelligent behaviour. It just studies a particular kind of computation and its realization by computers. This work is related to computational AI (section 2.2.2) although there is a difference: computational AI is concerned with intelligent behaviour, whereas computer science is not.

AI is not Engineering

Some people take AI to be an engineering activity, [Jones 90]. This view seems to be prompted by the fact that, within the Symbolic Paradigm, computer programs are built—AI is sometimes called clever programming. In [Ford 87] Ford compares AI with Software Engineering. He considers that both (AI and Software Engineering) can address the same problems but that, depending on the problem, AI's treatment will be more appropriate than that of Software Engineering.

In my view AI research is not simply an engineering activity; it is not a *clever* version of Software Engineering. AI research is concerned with investigating and understanding artificially created intelligent behaviour. Part of that understanding involves the engineering of artificial systems that display some kind of intelligent behaviour.

Investigations in classical experimental sciences, like chemistry and physics, often involve a large amount of engineering. For example, in astronomy, scientists spend a lot of time designing and engineering the necessary tools, e.g., telescopes. However, this doesn't make astronomy an engineering discipline concerned with how to design and build good telescopes. Researchers in AI also spend large amounts of time and effort building systems. These systems are the means to investigate the intelligent behaviour that researchers are interested in, and not some intended end product of it, as it is in engineering.

2.2.6 Recapitulation

I have presented AI as a science that studies intelligent behaviour in the artificial by means of experimentation. I have discussed this view in relation to other sciences and to various paradigms in AI. In general, work in Symbolic Functionalism (except Computational AI) together with Connectionism, and Robotic Functionalism can be characterised using this view. I have argued against understanding AI as a part of computer science or an engineering discipline. In the next section, I take a step further in considering AI as an experimental science and discuss the role of experiments in AI.

2.3 The Role of Experiments

One approach to science and the scientific method is based on ideas advocated by Karl Popper [Popper 68]. Popper's rather radical approach to science is also known as *Falsificationism.* According to Popper, observation in science presupposes and is guided by hypotheses that we must attempt to falsify. Hypotheses are proposed to be refuted by scientists as a result of observations and experimentation. However, Popper's radical falsifiability condition of hypotheses is too strict, mostly because straightforward and conclusive falsifications of hypotheses are not typically achievable in practice [Chalmers 82, chapter 6].

Lakatos' idea of the *research programme*, [Lakatos 74], provides a better understanding of scientific research. In a research programme hypotheses and theories are organised structures. The aim of experiments is then to test ideas and alternative directions for further development (as part of that structure) not just to refute individual hypotheses. Understanding from experimentation is used to develop coherent explanations and theories which have predictive power, that is, to extract general laws or principles, or to offer evidence for a hypothesis being tested.

For example, in Symbolic Functionalism some experiments involve building computer programs and testing them with respect to the intelligent behaviour they are intended to engender—chess playing, natural language understanding, or some kind of domain expertise, for example. Understanding from a series of experiments can then be used to offer evidence for the PSSH. Examples of this

kind of experiment are the General Problem Solver (GPS), [Newell & Simon 76], Soar, [Laird *et al* 87], and MYCIN, [Buchanan 88].

Not all experiments in AI are attempts to replicate some kind of intelligent behaviour. Some are to do with testing (computational) techniques or developing the necessary technology to support the performing of experiments to replicate intelligent behaviour. For example, Buchanan, [Buchanan 88, page 223], describes an experiment in which the use of meta-rules was tested to guide MYCIN's reasoning and make it more efficient. Whole areas of research (in this paradigm) are devoted to particular techniques such as search, constraint satisfaction, theorem proving, and proof plans which can also be found as headings in conference proceedings like ECAI92, [Neumann 92].

This thesis is mainly concerned with experiments performed to investigate intelligent behaviour. However, results of my investigations can be useful in the broader context of program understanding, both inside and outside AI (see section 7.5.5).

2.3.1 The Role of Programs in Experiments

The role of computer programs as part of experiments in AI is understood in very different ways by different practitioners in the field. In [Ritchie & Hanna 84] Ritchie and Hanna outline the lack of a clear way of carrying out research in the field. They do so by describing the different roles attributed to programs and how they contribute (or not) to the scientific development of research in AI.

Programs are sometimes understood as the means and ends of research. The aim in such an approach seems to be to demonstrate that programs that behave in a clever way can be built. This is what McCarthy called the "look ma no hands" approach. In this approach the researcher writes a program which usually does something no other program had done before (with sometimes impressive behaviour, or so it appears) and writes a paper saying so. Hayes called this approach "a sin of omission" because the researcher does nothing else but write a program [Hayes 75, page 15]. Ritchie and Hanna describe this kind of research as theoretically sterile because it does not develop principles nor does it clarify or define the real research problems.

Other times programs are supposed to implement some theoretical principles that the researcher develops. The problem here arises when programs are

presented as some kind of proof for the validity of a stated theory without further examination of either the theory or the program. In this approach the program (still complex and with apparent impressive behaviour) is sometimes accompanied by a statement of the performance it achieves. The problem with this approach, as Ritchie and Hanna point out, is that the written 'theory' may not correspond to the program in detail, the program may have been designed and built in a different way from the documented theory. The problem gets worse when the discrepancy is not accounted for, or even mentioned by the researcher. The theory is not subjected to examination except in so far as a program has been built. The existence of the program is presented as empirical justification for the theory. Even when the program is accompanied by a statement of the performance it achieves, it is not clear how that performance *is* achieved. Again, this kind of research has little to offer to scientific development. (See [Partridge & Wilks 90, chapter 6] for more on the role of programs in AI).

2.3.2 Programs and Experimental Procedure

We have just seen that it is not enough to present a program and some statement of the performance it achieves for it to be of some scientific value. However, as Buchanan points out, [Buchanan 88], much practice in AI research stops after a program is built:

> AI seems to contain a progression of steps from theorizing to engineering, from engineering to analysis, and from analysis back to theorizing. All seem to be important for progress. Upon examining individual pieces of research, however, we note that many researchers stop before completing the progression. Most work to date falls into either the theoretical or the engineering categories. Future progress requires additional work on analysis and generalization that is characteristically scientific.

2.4 Analysis in the Experimental Procedure

This thesis is concerned with investigating a way of supporting the analysis of AI programs. This analysis involves mainly two aspects: the structure and behaviour of a program. Newell and Simon, for example, stress these two aspects in the analysis of experimental programs, [Newell & Simon 76, page 36]:

> Each new program that is built is an experiment... Neither machines nor programs are black boxes... we can open them up and look inside. We can relate their structure to their behaviour... Inspection of the program in the light of a few runs reveals the flaw and lets us proceed to the next attempt.

2.4.1 Static and Dynamic Analysis

The static analysis of a program aims to account for the structure in the program that is responsible for the program's behaviour. The dynamic analysis provides a description of the kind and range of behaviour displayed by the program. These two aspects of a program, structure and behaviour, are strongly entwined, and their analysis, static and dynamic, respectively, are thus both necessary and complementary.

Although static and dynamic analysis are strongly related, they can be investigated separately. For example, Buchanan, [Buchanan 88], is interested in the dynamic analysis of programs. He advocates the need for controlled experiments for the dynamic analysis of complex programs. From these experiments data of the program's behaviour are collected and analysed.

The detailed static analysis of a complex program to establish its causal structure is a difficult task. One reason is that it can be extremely difficult to analyse a complex program in sufficient detail to establish what its essential causal structure is (see [Ritchie & Hanna 84] for a good example). This is particularly the case if the program was built without a clear understanding of what the causal structure might ultimately need to be to produce the intended behaviour. This is often the case in experimental AI programs, and as a result their analysis is often neglected.

This thesis is concerned with the investigation of a method to support the static analysis of AI programs. We will see in the coming chapters that results from dynamic analysis are used in this method for validation.

2.4.2 Static Analysis and Causal Structure of Programs

The static analysis of a program is strongly informed by its purpose. It is not the same to understand a program in terms of its connection to a database, the design decisions implemented in it, or how its efficiency is achieved. In the first case the understanding may involve finding the relationship between the database and program components. In the second case it can involve mapping resource exchange among modules in a program [Choi & Scacchi 90]. In the third case it might involve understanding how implementational issues can be used to provide greater efficiency.

However, these different kinds of understanding are not concerned with understanding the causal structure that is responsible for the intelligent behaviour the program displays. Investigating a way of supporting the analysis of AI programs to achieve this kind of understanding is the concern of this thesis. There are two important questions in this analysis: what is involved in understanding a program's causal structure that is responsible for the intelligent behaviour displayed by the program? What is this causal structure constituted of?

This causal structure can also be described in Newell and Simon's terms as the *further organization* of a physical symbol system. A fundamental concept of a symbol system is *designation* which is defined as, [Newell 80, page 156]: "An entity X designates an entity Y relative to a process P, if, when P takes X as input, its behaviour depends on Y". This means that for the purpose of process P, having (the symbol) X is the same as having (the designated entity) Y. Thus, the causal structure of a program is the further organization of a symbol system that gives rise to the necessary designations.

Representation is another term to describe this causal structure. Newell describes representation as another term for a structure that designates, [Newell 80, page 176]: "X represents Y if X designates aspects of Y, i.e., if there exist symbol processes that can take X as input and behave as if they had access to some aspects of Y".

In the context of engendering intelligent behaviour, what is represented is

usually knowledge, knowledge of the external world required to engender the intelligent behaviour being investigated. For example, in a chess playing program, it is most likely that there will be symbols designating the board, chess pieces, and in general, knowledge required to play chess. Thus, the causal structure is also described in terms of the knowledge representation, and associated reasoning aspects, that are necessary for the intelligent behaviour displayed by a program. This is stated very well in Brian Smith's *Knowledge Representation Hypothesis* [Smith 85]:

> Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.

Thus, understanding the causal structure of a program involves identifying the structures and the processes manipulating the structures. These structures and processes need to be interpretable as representing the knowledge required for the intelligent behaviour displayed by the program, and they have to be shown to be causal and essential in engendering this behaviour. This is usually formulated in terms of data structures and their operations [Newell 80, page 176]:

> Representation is also sometimes formulated in terms of a data structure with its associated *proper* operations. This view emphasizes the coupling of the static structure (what is sometimes called *the* representation) and the processing that defines what can be encoded into the structure, what can be retrieved from it and what are the transformations it can undergo with defined changes in what is represented.

A similar view on representation is presented in [Cummins 89]. Cummins stresses the fact that successful representation in computational systems depends essentially on the processes that have computational access to the symbols, and is not simply a matter between the symbols and their interpretation by the programmer. Thus, misrepresentation in programming has not only to do with data structures but with the combination of data structures and the way they are processed. It is often difficult to say whether unsatisfactory performance is

best improved by altering data structures (i.e., by finding a better 'knowledge representation') or by altering the processes that operate on the data structures.

Thus, this thesis is concerned with the investigation of a method to support the static analysis of AI programs. This static analysis is aimed at the identification of the program's causal structure, i.e., the essential data structures and their operations, and their representational role.

Static Analysis and Control

We have seen that to account for a program's behaviour it is necessary to identify the data structures and operations in the program. It is also necessary to identify the control mechanisms that organise these data structures and operations to produce the program's behaviour (on execution) at runtime.

Representation and control are both important for a complete understanding of the causal processes underlying and producing the program's behaviour. Their identification also raises complex issues of their own. This thesis is mainly concerned with identifying the data structures and operations in a program, rather than with the program's control. Note that it is possible to investigate representational issues without involving the program's control. However, the contrary is not true. To understand the control in a program it is usually necessary to have a prior understanding of the data structures and operations that are being controlled. Thus, investigating a way of identifying data structures and operations is a prerequisite to the investigation of issues of control. I will illustrate this point with an example in section 7.5.1.

2.4.3 The Role of the Researcher

One way of understanding a program is getting a description from the researcher who built it. Relying on the researcher to obtain a good description of how and why the program does what it does, poses a problem: what the researcher believes is in the program, and what is actually in the program (and how it gives rise to the observed behaviour) may be two different things. We can easily imagine

situations where aspects of a complex system might not be clearly understood by the researcher. This happened during research on the Edinburgh Designer System [Smithers *et al* 89] where the program's behaviour did not correspond to what was thought to be implemented, e.g., the system failed to recognise derivable inconsistencies from design decisions because combination of knowledge sources did not work as expected in the forward inferencing process (intended to support the designer). Another good example of the need to differentiate between what a researcher thinks is in a program and what is actually there, is given in [Ritchie & Hanna 84], and I will expand on it next.

2.4.4 An Example of Static Analysis

An example of the static analysis of a complex program is provided by Ritchie and Hanna in [Ritchie & Hanna 84]. In the early 1980s, Ritchie and Hanna attempted to do a rational reconstruction of Lenat's AM system. In rational reconstruction the idea is to reproduce the essence of the program's significant behaviour with another program constructed from descriptions of the important aspects of the original program [Partridge & Wilks 90, page 235]. AM is described by Lenat as a system which 'discovers' concepts and conjectures in elementary mathematics. The system does this by progressing from prenumerical knowledge to that of an undergraduate mathematics student, [Lenat 76].

Ritchie and Hanna never did the full rational reconstruction of AM that they originally intended. To reproduce the essence of Lenat's program, they performed an extensive static analysis. This involved looking at code from the program, as well as looking at Lenat's written accounts. They discovered discrepancies between published claims about the performance of the program and the techniques used in it, and the actual implementation. As a result of the analysis they questioned several aspects of the data structures and operations found in the system. It was not clear, for example, that data structures were as uniformly represented in the program as Lenat said, or that the control structure was the simple, uniform mechanism claimed. The effects of extra processing mechanisms on the program's behaviour was questioned, including whether part

of the real discovery work in AM was not in fact buried in unexplained procedures. Furthermore, they wondered if the mathematical discoveries of AM were really an important aspect of the work, and if they could be shown to be the direct result of the search procedure Lenat described, [Ritchie & Hanna 84, page 262]. Ritchie and Hanna concluded that the written accounts given by Lenat gave a misleading view of how his system worked, and that this obscured its real contribution and it could confuse other researchers in the field.

Just having a program whose behaviour is *believed* (by the programmer, at least) to be caused by processing appropriate structures is not enough. If our experimental investigation is to be effective we must also establish that this is indeed the case, and that the program's behaviour is *not* due, for example, to some aspect of the way it happens to be implemented—which can quite easily be, and often is, the case. For example, implementational details can take advantage of some language specific feature. Thus, good experiments are those that involve *not only* the design and building of computer programs, but also the analysis of the programs built. The program's static analysis involves showing that the program embodies a coherent system of data structures and operations, and that it is their accurate implementation that gives rise to the observed behaviour of the program.

2.5 A Framework of Levels for Experiments

When performing experiments we need to differentiate between the behaviour being investigated, the causal processes underlying and producing that behaviour, and how to create artificial systems that implement the processes and thus display the behaviour. Each of these three aspects require different understanding and descriptions. To distinguish them, I propose a framework of three levels of understanding—Knowledge Level, Symbol Level, and System Engineering Level. Identifying and describing the behaviour (Knowledge Level) is different to (and can be made independently of) identifying and describing the causal structure underlying the behaviour (Symbol Level), which is also different from an actual

implementation and its detailed description (System Engineering Level). These levels are necessary to distinguish different aspects of an experiment, but they are still all views of the same experiment.

2.5.1 The Knowledge Level

We need to have a way of describing the behaviour to be investigated in an experiment, a way of describing the knowledge and competence an agent needs to have to display some form of intelligent behaviour. This is nowadays described at a well known and accepted level in AI—the Knowledge Level, which was first proposed by Newell [Newell 81], and which I will use in my framework.

In Newell's terms, the behaviour of an agent depends on the agent's knowledge, where knowledge involves a notion of competence, a potential for action. At the Knowledge Level the system is the agent which has a body of knowledge, a set of actions, and a set of goals. To attain a goal, an agent selects and executes actions which are determined by processing knowledge. Thus, the behavioural law of an agent is the principle of rationality: actions are selected to achieve goals. For example, if an agent has knowledge of arithmetic and numbers, and it has as actions add and multiply, and its goal is to add two numbers, the agent will process its knowledge from which (if it can) it will select the add operation, and apply it to the numbers to achieve its goal.

Although the Knowledge Level is crucial in the experimental procedure, this thesis is not concerned with how to build good descriptions at this level. The ones provided for the experiments in this thesis (see chapter 8) are built as illustrations of what constitutes a Knowledge Level description, and they are useful for the discussion on the use of the framework for experiments in chapter 8, but they are not to be taken as examples of how to build good Knowledge Level descriptions.

Building Knowledge Level descriptions is, in general, a difficult task. Its investigation involves issues such as knowledge elicitation and formalization which are the concern of the epistemological programme [McCarthy & Hayes 69], and of the knowledge modelling community. The KADS (Knowledge Analysis

and Documentation System) methodology, [Wielinga *et al* 92], is an example of research aimed at providing a model of the problem solving behaviour required by a system. For a discussion on frameworks for studying expertise at the Knowledge Level see [Steels 90], and [Van de Velde 93] for a discussion on the methodological role of the Knowledge Level.

2.5.2 The Symbol Level

Applying the characterisation of science in section 2.1 to Symbolic Functionalism in AI, we can say that its overall aim is the development of abstract descriptions and theories about intelligent behaviour (created artificially). These abstract descriptions should be expressed in a way independent of, and without reference to, the particular form of their implementation, i.e., independently of the programs that display the intelligent behaviour under study. The description of the theoretical laws governing a certain form of intelligent behaviour is also different from the description of the behaviour itself, provided at the Knowledge Level. It is different because it provides a (internal) structure that the Knowledge Level is not concerned with (in Newell's terms). Thus, this description should be provided separately from the descriptions at the Knowledge Level.

I propose the Symbol Level as the level at which this kind of abstract description is provided. It is a level abstracted from implementational details of individual programs but which captures the further organization of the symbol system that engenders the intelligent behaviour described at the Knowledge Level. It is at the Symbol Level that we describe a program's causal structure, i.e., the data structures and their operations identified, during the static analysis, as responsible for the observed behaviour (section 2.4.2). What is not described at this level is the particular implementation of these data structures and operations.

This Symbol Level is to be distinguished from others with the same name that can be found in the literature. For example, Newell also talks about a computer Symbol Level. In fact, he says that the Knowledge Level was derived as the result of splitting the Symbol (or program) Level in two. From the original Symbol Level Newell takes one aspect to create the Knowledge Level, but maintains

the rest as the Symbol Level. The remaining Symbol Level is thus anything and everything between the computer Logic Level and the Knowledge Level: programming languages at different levels (from low level programming languages like assembler to high level ones like LISP) different kinds of programs (from operating systems to programs playing chess) computational techniques, (e.g., theorem provers, or search techniques) or system architectures (e.g., blackboard architecture). The problem with this symbol level is that it is too broad to be useful when designing and executing experiments in AI research. It does not distinguish programs from their abstract descriptions which I think is an important distinction.

Another Symbol Level that can be found in the literature is proposed by Brachman and Levesque, [Brachman & Levesque 86], as part of their framework to build Knowledge Base Management Systems (KBMS). Their framework is also constituted of a Knowledge Level, a System Engineering Level, and a Symbol Level. The last one involves describing data structures and algorithms necessary to process and view knowledge bases (e.g., locking, parallelism, and management of secondary storage). Their Symbol Level describes how the machine views the system, and they say that in a programming context it describes the object code that is executed after compilation (see footnote six in [Brachman & Levesque 86]). This is a much lower level view of a system to the one I am proposing to call Symbol Level. In the Symbol Level I propose a system is viewed abstracted from implementational details, contrary to the machine's view that constitutes Brachman and Levesque's Symbol Level.

2.5.3 The System Engineering Level

The description of the implemented artificial systems is also at a different level of understanding from the previous two. This is a description of each particular implementation of the data structures and their operations (described at the Symbol Level), and that thus displays the intelligent behaviour investigated (described at the Knowledge Level). Artificial systems in this paradigm are created by building computer programs, so describing an implementation involves

describing the program built. The implementation of a program depends on how it is built and what it is built with: the programming language (e.g., Lisp, Napier88) machine (e.g., Lisp machine, personal computer) and the programming methodology used (e.g., prototyping, top-down). These are aspects that influence the implementation of a program but that do not influence the previous two levels of description. This is why, the implementation should be described at its own level.

I propose the System Engineering Level as the level at which the description of the implemented program is provided. We will see in the next section that experimental AI programs are usually built without a clear specification and by prototyping. The description at this level includes the program itself (the code of the program is in itself a written account of the implementation) and any documentation of the way it has been built, i.e., initial decisions and changes on those decisions (that the builder is conscious of) during the building process, and records of prototypes built on the way to the final program.

This level should not be confused with Brachman and Levesque's System Engineering Level, [Brachman & Levesque 86], which focuses on organisational aspects of a knowledge base, and on information-abstraction mechanisms that help a designer to manage the design of a knowledge base (e.g., successive refinement by specialization and version management). In a programming context, they view their System Engineering Level as dealing with features of high-level programming environments which would also be part of the System Engineering Level I am proposing for performing experiments. However, the level I am proposing is broader than that of Brachman and Levesque. It includes other aspects of program building, such as things that Brachman and Levesque view as being at the Symbol Level (e.g., locking and parallelism).

Research into the System Engineering Level involves developing languages and environments for AI programming, programming methodologies, techniques, and mechanisms that have to do with building computer programs. I will expand on this level in section 3.2.

2.5.4 A Framework of Three Levels

I propose a framework of three levels with which to understand an experiment in AI research.[5] These levels are views or levels of understanding of the same thing, they are all part of the understanding of an experiment. They are not intended to suggest that there is a hierarchy of importance involved. The aim of this framework is to separate out the different kinds of issues that have to be resolved when performing an experiment to create some form of intelligent behaviour. The proposed framework is described in terms of these levels and the way they are related:

- The *Knowledge Level.* This level treats knowledge as a competence notion, a potential to act. It specifies what kind of knower a system needs to be to behave intelligently, and what knowledge is required to be such a knower. This level presents a description of knowledge required to create the behaviour investigated in the experiment.

- The *Symbol Level.* This level is concerned with the description of the data structures and their operations, the 'further organization', that are responsible for engendering the behaviour specified at the Knowledge Level. That is, it provides an accurate description of their interpretation as the knowledge representation required, and thus, of their essential role in engendering the behaviour.

- The *System Engineering Level.* This level is concerned with describing the particular program implemented in the experiment, a program that, by accurately realising the data structures and operations described at the Symbol Level, displays the intelligent behaviour described at the Knowledge Level.

The Knowledge Level is central to the understanding of what the system is intended to do, what kind of intelligent behaviour we want to investigate with the

---

[5]This framework was first proposed in [Bernaras & Smithers 90].

particular experiment we are designing. There is no structure at the Knowledge Level. It provides the means to rationalise the behaviour of a system, from the standpoint of an external observer, in terms of goals, actions, and the knowledge needed to select these. It therefore acts as a specification of the external behaviour and as a working hypothesis of the goals, actions, and knowledge that an agent, that behaves in the specified way, will have. It is a *working* hypothesis in the sense that this aspect of the Knowledge Level description may be modified during the building of the program. This can happen as more is discovered about what it takes to produce the specified behaviour. The Symbol Level, on the other hand, acts as a specification of what symbol structures and processes must be realized to engender the behaviour specified at the Knowledge Level and which can be interpreted in terms of goals, actions, and knowledge also described at the Knowledge Level. (See section 8.3.2, page 250, for an example of establishing such an interpretation.) A crucial difference between the Knowledge Level and the Symbol Level is that the former is behaviour oriented, whereas the latter is system oriented. Finally, the System Engineering Level describes the implementation of the symbol structures and processes employing computational data structures and mechanisms.

Performing good experiments involves obtaining good descriptions at these three levels. We will see in the next section (also in section 3.2) that in AI research it is not usually possible to provide an accurate Symbol Level description prior to the building of the program. It is as a result of the program's static analysis that such an accurate description can be provided. This is why this thesis is concerned with investigating a method to help researchers performing the static analysis of their programs, an analysis aimed at providing an accurate Symbol Level description of the experiment.

This framework can be related to Buchanan's characterisation of the steps followed in AI. We saw in section 2.3.2 that they involve initial theorizing, engineering, analysis, and back to theorizing. The Knowledge Level describes the initial theory about the intelligent behaviour investigated. The System Engineering Level describes the engineering performed to test the theory. The Symbol Level describes the causal structure discovered as a result of the analysis, and that in itself constitutes a theory of the structure, the further organization, required to engender the behaviour investigated. Thus, the final theory includes the description of a certain behaviour and the knowledge and causal structure required to engender it.

2.5.5 Other Frameworks

Other levels of understanding can be found in AI and other disciplines. Here I mention two examples of similar frameworks that can be found in the literature: a framework of three levels used for building databases, and Marr's three levels for doing computational vision experiments.

Levels in Databases

A framework of three levels (or views) is used for designing and building databases and database management systems (like Brachman and Levesque's for KBMS): *Conceptual Level*, *Logical Level* and *Physical Level* [Parsaye *et al* 89].

The Conceptual Level is concerned with representing an abstract view of the physical data (content) and how it is stored (structure) in the database. It offers an abstraction of the database for the users. This can be seen as a similar level to Newell's Knowledge Level in that it somehow specifies the abstract behaviour of the database, although in a more restricted way, i.e., it specifies structure which is not the case in the Knowledge Level. The Logical Level is concerned with describing the logical structure of the database. It involves precise definitions of properties that must hold at all times, i.e., access and security requirements, and that are required to realize the description at the Conceptual Level. This is similar to the Symbol Level in that these properties are abstracted over the particular database implementation, however, the Symbol Level also includes structure which is at the Conceptual Level. Finally, the Physical Level addresses specific implementation issues—methods for data access and management, file organisation, etc. This level is concerned with the implementation of the logical structure described at the Logical Level, and that is required to realize the behaviour described at the Conceptual Level. This is a similar level of abstraction to the System Engineering Level, although what is implemented in them is very different.

Marr's Levels for Computational Vision

Marr, [Marr 82, page 24], proposes three levels at which to understand machines performing information processing tasks. He calls these levels, *Computational Theory Level, Representation and Algorithm Level,* and *Hardware Implementation Level.*

Marr's levels are proposed in the context of computational vision. This means that at the Computational Level (where the goal of the computation is defined together with why it is appropriate and the logic of the strategy with which to carry it out) the underlying task is to derive properties of the world from images of it, not to describe a potential for intelligent behaviour as in the Knowledge Level. At the Algorithm Level the algorithm for the strategy and the representation for the input and output of the theory are specified, and at the Implementation Level the algorithm is implemented. The problem with these last two levels is that there isn't a very clear distinction between them. Marr says that the choice of an algorithm (at the Algorithm Level) depends on the machinery it will be implemented in (Implementation Level). As Newell says in his discussion of the computer levels, [Newell 81], a level is defined without reference to a level below, although if it is realizable, it will be reduced to it. Thus, the choice of algorithm should only be made on the basis of its adequacy to the task, and abstracted from implementational issues. This is an important distinction between the Symbol Level and System Engineering Level in my framework for experiments.

2.6 Using the Framework for Experiments

The experimental methodology followed in AI research can be usefully described using the framework of three levels just presented. This is also the case for the methodology followed when building AI application systems. Using the same framework in both, research and application, helps understanding their differences in methodology and aim. As will become clear in the coming sections, the work on understanding programs done in this thesis is well suited to the experimental methodology, but not so much to the application building methodology.

2.6.1 Experimental Methodology

I describe the experimental methodology as a three step process, see figure 2–1(a).[6] The arrows in figure 2–1(a) reflect the sequence in which the steps are performed. The first step consists of deciding and describing, at the Knowledge Level, what intelligent behaviour to investigate. However, the description at the Knowledge Level does not tell us what data structures and operations are required to engender it. As a consequence, we do not have a well formed and complete specification for the programs we build, though we usually have some idea of what it could be like and how it might be built. It is important to realise that this initial Knowledge Level description can get modified during the process of building the program. This process is likely to improve our understanding of the behaviour being investigated, the knowledge required for it, and thus, their description.

Thus, the second step is to try to build a program to realise this behaviour. This step is carried out at the System Engineering Level (arrow going from the Knowledge Level to the System Engineering Level in figure 2–1(a)). Building a program that displays this behaviour doesn't tell what is the program's essential causal structure, not at least directly.

---

[6]This methodology was first presented in [Bernaras & Smithers 90].

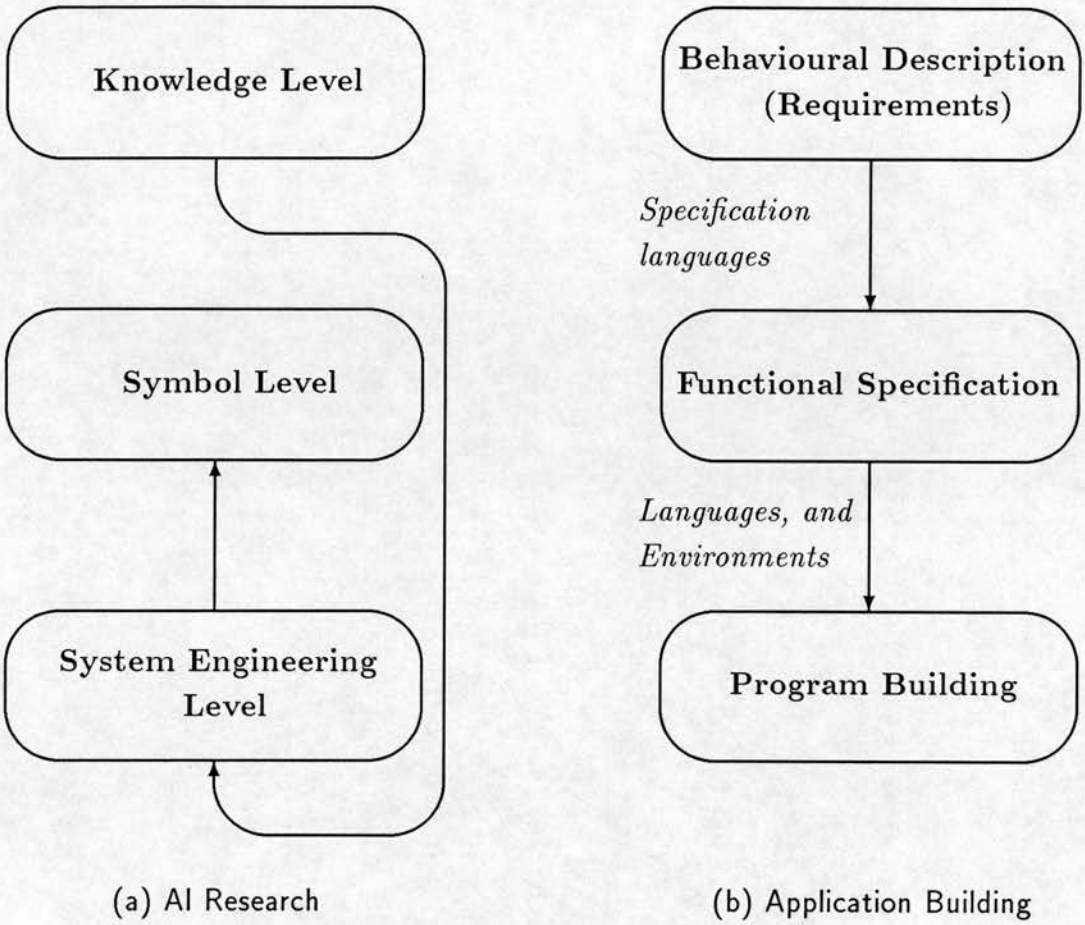


Figure 2–1: AI research and application building methodologies.

The third step in the process (arrow from System Engineering Level to Symbol Level in figure 2-1(a)) has as inputs the program (or programs) built, the current Knowledge Level description (which may be different from the initial one) and the program builder's knowledge gained from the experience of implementing, debugging, and testing the program (or programs). In this step, we analyse the program to try to discover, and to describe, the data structures and their operations that the program implements and which can be taken as engendering the Knowledge Level behaviour being investigated. The result of this analysis is the Symbol Level description for the experiment. The analysis method to be presented is designed not to depend on any knowledge of preconceived ideas the programmer may have about the implementation of the program. This is important since this knowledge can be an unreliable basis for understanding how the program actually works.

For this methodology to work we must be able to understand the programs we write in a way that enables us to subsequently extract from them and describe their essential causal structure.

2.6.2 Methodology for Application Programs

By contrast, when building application programs, the aim and thus the methodology followed are different, see figure 2–1(b). To contrast the experimental methodology with the application building methodology, I illustrate how the framework of three levels can be seen as also forming the basis of a design and construction methodology for knowledge-based system (KBS) applications.

In the application building case, the process can again be described in terms of three steps. Starting from a description of the problem to be solved, the program builder first constructs a description of the required behaviour—a Requirements Description. In the second step—Functional Specification—the Requirements Description is translated into a set of formal specifications that define the functionality (in terms of top level data structures and operations) of the program required to deliver the desired behaviour (arrow from Requirements to Functional Specification in figure 2–1(b)). Specification languages can be useful in this step. Finally, in the Program Building step, the Functional Specification is used as the input to the process of designing and building a program to be delivered, and that realises the solution to the application problem (arrow from Functional Specification to Program Building in figure 2–1(b)).

In other words, when building application programs in AI we should be following the same methodology as for other kinds of application software building. The Knowledge Level is understood as the Requirements Description, the Symbol Level as the Functional Specification for the program, and the System Engineering Level as the Program Building.

2.6.3 Why Different Methodologies?

To understand the difference between experimental methodologies for AI research and the application building methodology it is important to understand their different aims. Application programs are end products to be delivered as utilities. In application, the end product has to be clearly understood and specified before it is finally built. The builder and the user have to agree on what the end product

is going to do. Reaching an agreement may involve building and evaluating an early prototype to establish clear requirements, but that prototype is part of the requirements description phase, not part of the application building.

In contrast, programs for research are the means for experimentation. They are vehicles for learning about and understanding a phenomenon. There is no product to be delivered, no user to deliver it to, and therefore, there is no problem in changing or adding to the initial requirements (which are usually incomplete and perhaps inconsistent) as understanding of the phenomenon develops.

The difference in aim is reflected in differences in other aspects like the life cycle of the programs, their maintainability, security issues, user friendliness, etc. The life cycle of an application program involves the specification of requirements, design, implementation, and testing of a final product, as well as its maintenance. Research programs don't usually require maintenance because the program built is not a product to be delivered and used as an utility. It may even be the case that after we have learned what we wanted by building, testing, and analysing a program, the program is thrown away.

Other aspects of the life cycle remain the same, or at least similar. Building research programs also involves designing and implementing a system, and it involves an analysis phase of the resulting program. It is interesting to note that part of the analysis can involve similar kinds of testing to those in application programs (e.g., checking consistency and completeness of the knowledge base).

2.7 Conclusion

AI is an experimental science concerned with studying intelligent behaviour in the artificial. Experimentation in the Symbolic Paradigm involves not just designing and building computer programs, as is often the case, but also analysing them afterwards. This analysis should provide a description of the program's causal structure.

No framework in the literature adequately covers these aspects of performing experiments. A first step in the investigation of this thesis involved proposing a framework for experimentation based on three levels: Newell's Knowledge Level, Symbol Level, and System Engineering Level which are related to other frameworks that use the same names (Brachman and Levesque's levels for KBMS) or frameworks in other contexts with similar levels (Database levels and Marr's levels for computational vision). I have described the experimental methodology followed in AI research in terms of these three levels, including the nature of the programs' building process. The next chapter will address the problem of analysing AI research programs given the nature of their building process, together with the approach proposed in this thesis to tackling it.

Chapter 3

The Analysis of AI Research Programs

In the previous chapter I presented AI research as an experimental science. I discussed the role of programs as part of experiments in the Symbolic Paradigm, and the need to analyse them to provide an accurate description of the program's causal structure. I presented a framework of three levels at which AI experiments can be understood—Knowledge Level, Symbol Level and System Engineering Level. I also presented the experimental methodology followed in AI research in terms of these three levels. In this chapter I will discuss the problem of analysing AI research programs given the nature of their building process, and I will present my approach to tackling this problem. This will be followed by a re-statement of my thesis, and a brief sketch of the analysis method developed during its investigation. The last section will review other work on understanding programs, and their relation to my work.

3.1 Understanding Programs: The Problem

This thesis is concerned with investigating a way of supporting the static analysis of AI programs as part of the experimental procedure. Two aspects need to be considered in the analysis: the purpose of the analysis and the nature of the programs being analysed.

We have seen in chapter 2 that the purpose of the analysis is to help in the identification of the data structures and operations in a program and their

description at the Symbol Level. The nature of a program is affected by its building process, and thus, it is important to consider it during the analysis. As we will see in the next section, looking at programs and their building processes involves looking back into the System Engineering Level.

3.2 The System Engineering Level

Here I examine how programs are built in AI research in terms of the initial information available, and the programming methodology used to build them. The initial information refers to whether a full specification of the problem is available prior to the program construction. This is important because the analysis of a program built from a full specification is different from the analysis of a program built from a few ideas of what the functionality of the program might be and how it might be achieved. In the first case the analysis will have available not only the program but a well defined specification of it. In the second case the analysis will be mainly based on the program.

During the building process, the initial information about the program influences the choice of the appropriate programming methodology. With a full specification, a top-down or stepwise-refinement methodology can be used to build the program. In the case of only having very weak specifications, a prototyping or bottom-up methodology is more appropriate.

The analysis is also affected by the programming methodology used: the analysis of a program developed by a stepwise methodology could be performed by going backwards in each step of the stepwise-refinement until the result of the analysis can be contrasted with the original specification. The analysis of a program developed by prototyping is not that clear, and the result can only be contrasted with the ideas and intentions of the researcher, not with a clearly stated specification.

3.2.1 Specifications and Prototyping

The aim of a scientific enterprise is to understand certain phenomena. Understanding is achieved by investigating phenomena and formalizing results from the investigation. A formalization is a kind of specification of what must happen for a phenomenon to occur. This kind of specification is an important achievement in science, and in our particular case, in AI research. In programming, a specification is also a formal description of what a program does. It contains a set of pre-conditions and a set of post-conditions that describe the initial and final states, and a frame that lists the variables whose values may change. A specification in programming can be understood as: if the initial state satisfies the pre-conditions, then change *only the variables listed* in the frame so that the resulting final state satisfies the post-conditions, [Morgan 90, page 6].

In AI research, specifications (a frame that lists the variables that may change) are not usually available before a program is built. In fact, this is a kind of information researchers look for: which variables affect the process being investigated and how. Building a program is the means by which a researcher seeks to understand a process, the variables that affect it, and how they affect it. Therefore, a specification in computational terms is not (and typically cannot be) obtainable *a priori.* The building process is mainly based on the researcher's ideas of what the variables may be, and how they might affect the process thought to be implemented in the program.

As a result of the lack of clear specifications, programs in AI research are not (and typically cannot be) developed following a top-down programming methodology. Steps, in a top-down approach, go from a specification in a high level specification language to a final detailed program written in a programming language. By contrast, programs in AI are developed following the so-called prototyping methodology which does not require complete specifications. Building prototypes or 'tries' introduces an element of communication and feedback. These are important aspects of the learning process on the way to a final program. Thus, programs in AI research are typically built without clear

specifications and following the prototyping programming methodology. The next section presents a taxonomy of prototyping based on [Floyd 84].

3.2.2 On Prototyping

A prototype is a learning vehicle. As such it provides more precise ideas of the phenomenon under investigation, and how to create it in a program. Prototyping can be characterised as a four step process: functional selection, construction, evaluation, and further use [Floyd 84]. The functional selection refers to the choice of functions which the prototype should exhibit. Construction refers to the effort required to build the prototype. Evaluation is the decisive step in prototyping since it provides feedback for further development, and the experimental process. Further use depends on the evaluation, i.e., on the experience gained with the prototype: if the prototype serves merely to learn about some particular aspect or method then it may be thrown away afterwards, but not if it is used fully or partially in the next prototype.

Three broad approaches to prototyping can be distinguished on the basis of the goals to be achieved with the use of prototyping [Floyd 84]—exploratory, experimental, and evolutionary. *Exploratory* prototyping focuses on the exploration of ideas and computational features or techniques that may be useful to obtain a required behaviour in a system. In an exploratory approach the focus is on the ideas developed while building the program, not on the program itself. This kind of program is usually thrown away afterwards.

Experimental prototyping focuses on proposing a solution to obtain a system that displays some required behaviour, not on testing individual ideas or techniques. It simulates a proposed solution. There are different kinds of experimental prototyping: *full functional* simulation is based on a prototype exhibiting a full range of functions to be available in the proposed solution, whereas *partial functional* simulation can be used to test a hypothesis about a part of the proposed solution. This kind of prototyping is mostly used in preliminary stages of the design, and it is also likely to be thrown away afterwards.

Finally, *evolutionary* prototyping (sometimes called *versioning*) is concerned with the development of an actual system. It does this by developing versions of a solution. It breaks down the linear ordering of development steps and maps it into successive development cycles. Depending on the degree of this decomposition we can distinguish two kinds of evolutionary prototyping (EP)—incremental and evolving. Systems built via *incremental* EP are those built gradually in a process of learning and growth. The main idea here is that complex problems are best dealt with by a stepwise extension of the solution. Systems built via *evolving* EP are those built within a dynamic and changing environment; prototypes evolve to accommodate subsequent, even unpredictable changes in the environment.

3.2.3 Prototyping in AI

Prototyping is widely used in AI; not only in research but also in application building. Many times these two areas of AI are confused, and I believe that the use of prototyping in both areas contributes to this confusion. I have already discussed methodological differences between research and application in AI in section 2.6.3. Here I will further add to their distinction by discussing how the different kinds of prototyping are used in these two areas.

Prototyping in AI research

In section 2.3 I discussed two kinds of AI research experiments: those performed to recreate some form of intelligent behaviour, and those performed to develop and test techniques. It is difficult to delineate clearly the boundaries between the different kinds of prototyping and which one is best suited for the different kinds of experiments. One experiment to recreate some kind of intelligent behaviour may involve mainly incremental prototyping as the way to obtain a system (a chess playing program for example) but exploratory prototyping may be used to investigate a new technique or algorithm (a new search method, for example); partial simulation (experimental prototyping) can be used to investigate if an existing technique proposed for one kind of system will produce acceptable results within another kind of system. An example is the use of a

truth maintenance technique (TMS) in a design support system, [Logan *et al* 91], rather than in a diagnosis system (the context in which most work on TMS is done, [deKleer & Williams 87]). Another example, at a much bigger scale, is the application of a whole architecture for general intelligence, like Soar, to different kinds of problems (see [Laird *et al* 87] for a description of Soar, and specially page 3 for a set of such problems). Having said this, here I will present the more typical prototyping methods in these two types of experiments.

Bundy, [Bundy 90, page 221], considers exploratory prototyping as the major methodology in computational AI. He describes the programming process in basic AI as follows: first, the researcher chooses some task which has not been modeled before, and then, by a process of trial and error programming, he or she develops a program that performs this task. Ritchie, [Ritchie 91a, page 2], describes the computer as a scratch-pad on which to try out ideas. Both descriptions can be equated with exploratory prototyping.

Experiments to recreate some form of intelligent behaviour usually involve building large and complex programs containing several functionalities. Incremental prototyping seems to be the most appropriate way of developing such systems. It gives the opportunity to develop a complex system in a coherent way by stepwise, revisionist extensions. This doesn't mean that complex programs are not built when a new technique is being explored. It is, nevertheless, less likely because the functionality of the program is restricted to that of the technique being tested.

An example of the use and benefits of incremental prototyping to develop complex systems comes from the development of the Edinburgh Designer System (EDS) of which seven incremental prototypes or versions were developed as part of the 'Design to Product' project [Smithers *et al* 89]. The Castlemaine project, [Smithers *et al* 92] (an attempt to apply the EDS architecture in the domain of drug design) is another example of the use of incremental prototyping: two consecutive prototypes were built incrementally.

The Prototype Analysis Method I have developed is intended to help in the

analysis of programs built by incremental prototyping, that is, it takes into account the versions built on the way to the final system.

Prototyping in AI Application

Prototyping is also applied in application building in AI (e.g., knowledge-based application systems or Expert Systems) and also in Software Engineering. There is a lot of debate about whether AI application programs can be developed following the classical software development methodology or whether they need a new, different methodology. Some authors consider that the steps and building cycle in AI application and Software Engineering are equivalent, only the details change, [Rauch-Hindin 88]. Others believe that although the classical top-down method should be used when possible, there are many problems in AI applications for which it is not adequate. For these, they propose what they call an exploratory programming methodology: specify-(re)code prototype-explore-freeze and optimize [Kreutzer & Mckenzie 91].

A more radical position views Expert Systems development as requiring a non-traditional development cycle based on early prototyping and incremental revision of code, [Luger & Stubblefield 89, page 299]. This position is also taken by Partridge, [Partridge 86], who advocates a new methodology for AI software development, the RUDE methodology (run-understand-debug-edit) that when performed properly, he says, becomes incremental development of a program.

A more conciliatory view between AI application and classical software development is that presented by Fikes, [Fikes 90]. He views prototyping as useful at early stages of a knowledge-based system application project. He considers that exploratory prototyping can help to determine a realizable detailed set of specifications of the functionality, knowledge and processing methods required to perform the task. This is a similar position to that held by [Floyd 84] for very complex Software Engineering projects. Floyd views prototyping as a component of the application software development methodology. As such, different kinds of prototyping can be used in each step of the classical development method. For example, exploratory prototyping can be used to enhance requirement and

functional analysis, experimental prototyping can be used in any step after the initial specification has been written, and incremental prototyping can be used during implementation (see [Floyd 84] for more on this view of prototyping).

I believe that prototyping (of the various kinds) in AI application typically has to be used within the classical top-down methodology (the use of this methodology for AI application has already been discussed in section 2.6.2). By contrast, prototyping has a different role in AI research as it is part of a different methodology, the experimental methodology (already discussed in section 2.6.1). The same kinds of prototyping can be used in both research and application but their purposes, and thus their use as part of different methodologies, differ.

Despite the different roles of prototyping, in section 7.5.5, I will discuss how results from my investigations are likely to carry over to other areas of AI and Software Engineering where prototyping is widely used.

3.3 Approach to the Analysis of Programs

In the previous section I have shown that the kinds of programs I am attempting to understand are those built without clear specifications and by incremental prototyping. In this section I present my approach to tackling the analysis of these kinds of programs. It involves their static analysis to identify specializations of data structures and their operations. The analysis is performed by a process of program transformation where results from dynamic analysis are used for validation, and where prototypes on the way to the final system are also analysed.

3.3.1 Operations on Data Structures

To understand what particular data structures represent, or, more usually, to confirm that they actually do the representational job that the program builder supposes them to do, we need to understand how they are processed (see section 2.4.2). In other words, to uncover a program's essential causal structure we need to have a coherent definition of all the data structures, and all the operations

that operate on each data structure in the program. Thus, my approach to the analysis is to obtain a uniform description of each data structure and its operations.

3.3.2 Specializations

Data structures are usually specialized by the program builder to describe the patterns used in the program to represent particular aspects of the domain. For example, the data structure list can be specialized to represent a list of books or a list of houses. Also, each specialization of a data structure is likely to be operated on in a different way, e.g., some of the operations that apply to a list of books are not likely to be applicable to a list of houses. During the analysis, it is important to identify these specializations. They provide a clearer semantic interpretation of the data structures and their operations, thus, helping to understand their representational role in the program.

These specializations are described by the types defined by the program builder. Identifying the specializations then involves identifying the different types in a program. It is for this reason that I call these specializations *user-defined types*, and the operations are those applied to each user-defined type. Thus, a user-defined type is an abstract notion used to describe any particular specialization of a data structure, or combination of data structures, of a type that can be formed in the programming language used. This notion is therefore not tied to particular language constructs, but, of course, particular instances of user-defined types will reflect the constructs available in the language used. (See section 6.1.1, page 171, for a more detailed discussion of the nature of user-defined types.) The analysis thus aims to identify the user-defined types and all the operations applied to each of them to understand their representational role in the program.

An interesting issue in this respect would be to consider the combination of user-defined types and their operations as patterns and to investigate the possibility of determining which patterns might be conventionally used in experimental AI programs. This is usually possible when analysing student programs in tutoring systems for example (see section 3.6.3) or conventional software systems (see section 3.6.5) where the same or similar tasks are performed repeatedly, and/or where there are specifications that help in determining possible patterns in a program. Identified patterns can then be used in these areas to perform automatic program recognition (see section 3.6.7).

Experimental programs are usually innovative in some respect, i.e., a new

task, or a new way of performing a task, is investigated. It is not clear that there are patterns identifiable as conventionally used in experimental AI programs in general (except perhaps for some restricted domains or techniques). Furthermore, even if there were some conventional patterns, experimental programs incorporate many user-defined types and operations that are unconventional, and their identification would still remain a problem. Thus, this thesis does not attempt to investigate which patterns of programming might be identified as useful in experimental AI programs. Rather, it investigates a way of supporting a researcher in identifying all the user-defined types and operations in his or her program.

3.3.3 Abstracting from Implementational Details

To understand what a particular user-defined type represents, implementational details, which are not essential for the program's causal structure, need to be abstracted away. The issue of abstracting from implementational details is thus tied up with the question of representation, that is, with what symbols stand for. Getting such a description straight from a program is very difficult. We will see in chapter 4 that it is in this process of abstraction that Software Engineering abstraction constructs are useful.

3.3.4 Program Transformations

In my investigations, achieving higher degrees of abstraction is the aim of the transformations performed in the program: a program is transformed from a form dominated by implementational and language dependent details to a more abstract form which hides many of these details, where language dependent refers to the constructs available in the language used. At the same time, this form offers a more uniform description of the user-defined types used by the program and of the computational operations performed on them and thus, a form from which a Symbol Level description can (more easily) be obtained.

Therefore, the approach taken in this thesis for the analysis of AI research programs is a process of incremental program transformation. This process is

at the same time a process of abstraction and specialization: abstraction from implementational details, and specialization of user-defined types and operations. We have seen that they are both important to support the uncovering of a program's essential causal structure.

3.3.5 Connecting Static and Dynamic Analysis

As a result of the transformation process a new more abstract but still executable program is obtained. The transformations, and the structure identified in the process, need to be validated. Having an executable form of the program is very important because validation can then be done by running the transformed program, and ensuring that it behaves in the same way as the original program. A test set taken from the dynamic analysis of the original program can be used for this purpose. In this way, dynamic analysis is used to complement the static analysis of a program.

3.3.6 Analysing Prototypes

Usually prototypes are thought of as the means to get to a final system. Often early prototypes of a program are forgotten, and even deleted, once the final system is developed, [Buchanan 88, page 224]:

> Too seldom are there records of initial implementations that failed to achieve desired performance, and almost never are there records comparing discarded and saved methods or written notes on why methods were changed.

The analysis of a final program involves as many difficulties as its building process. Sometimes even more because of the tediousness of the analysis task, quite different from the excitement of its development. Furthermore, analysing *only* the final program may not provide enough information for developing a good understanding of its operation. Prototype versions of the program are important in reaching the final system. I therefore believe that understanding them and their relationships is important to understanding the final version of the program. By taking into account previous prototypes, the analysis is likely to provide insight

into the researcher's development of ideas during the development of the program, thus helping to obtain a better understanding of the final system.

This is why the analysis of prototypes plays an important role in the understanding of a system. Incremental prototype programs reflect new or changing ideas developed by the researcher. Their analysis can help to understand better the changes in the program, and thus the final system, based on a better understanding of the process followed by the researcher. This does not mean though that the analysis is easier. The analysis process is likely to be more complex, and more work is involved in analysing several prototypes than just one. Nevertheless, I believe that the result is a clearer description of the causal structure of the final program, as well as possibly identifying the causes of failures or inadequacies of previous prototypes which, as Buchanan points out, is important in an experiment.

3.4 A Method for Static Analysis

In the previous sections I have discussed the problem of analysing AI research programs and my approach to tackling it. The conclusion from this discussion is that the kinds of programs I am attempting to understand are built without clear specifications and by incremental prototyping. Understanding programs in this context involves their static analysis to identify specializations of data structures and operations. What is known prior to this analysis are the current Knowledge Level description, the series of implemented incremental prototypes, and the program builder's knowledge gained from the experience of building the programs. The analysis is then performed by a process of program transformation where results from dynamic analysis are used for validation and where prototypes on the way to the final system are also analysed. The result of the analysis is the Symbol Level description for the experiment.

In this section I will re-state my thesis and I will introduce the analysis method I have developed during its investigation, the Prototype Analysis Method. Although the method will be described in more detail in chapter 7, here I will briefly explain how it relates to the approach taken and the Software Engineering abstraction constructs used in it. This brief explanation gives an idea of how I have investigated my thesis, which is this:

The use of abstraction constructs developed in the area of programming language research in Software Engineering can help to understand better experimental computer programs in AI research.

The Prototype Analysis Method (PAM) has been developed to investigate the use of certain Software Engineering abstraction constructs to help the process of understanding AI research programs. It is a static analysis method which both accepts the incremental and weakly specified nature of AI research programs and makes their analysis practical (but not necessarily easy). The word 'prototype' in the name of the method is there to indicate that, as part of the analysis, each incremental prototype program that is built on the way to the final program is analysed to see how it performs the way it does, and how it is structurally related to the other prototypes, and to the final system.

The method helps to abstract away much of the implementational detail of a program to reveal its essential causal structure. It is in helping to abstract away implementation detail from a program that I see the use of modern Software Engineering developments as being important. Software Engineering research is concerned with the formalizing and facilitating application program construction. Part of it involves research in programming language design, which provides abstraction mechanisms and computational constructs, such as strong typing, persistence, abstract data typing, and polymorphism. These constructs are intended to offer more abstract levels of programming, that is, to help bridge the gap between specifications and programs. We have seen that building application programs is very different from building experimental AI research programs, both in aim and methodology. Nevertheless I believe that these abstraction constructs can be useful in analysing experimental AI programs (as opposed to building them). As a result of the transformations of a program (to abstract away implementational detail) a new more abstract but still executable program is obtained. The transformations are then validated by running the transformed program using a test set taken from the dynamic analysis of the original program.

We saw in section 2.4.2 that understanding the causal structure of a program involves identifying the structures and the processes manipulating these

structures, that these structures and processes need to be interpretable as representing the knowledge required for the intelligent behaviour displayed by the program, and they have to be shown to be causal and essential in engendering this behaviour. The data structures and operations in their new form are thus interpreted in terms of the ontology of the problem domain or the problem solving method used. In this way, this analysis method is designed not to depend on any knowledge of preconceived ideas the programmer might have about how the program works.

3.5 Conclusion

Programs are built as part of the experimental methodology followed in AI and understanding these programs is the concern of this thesis. The method I have developed for this purpose is unique in that no other method that I know of tackles the understanding of programs in the context of experimental AI or the use of Software Engineering abstraction constructs to support the static analysis of programs built from weak specifications and using incremental prototyping.

However, there is a lot of research related to understanding written programs in Software Engineering and AI. The development of the method proposed in this thesis has involved looking at various aspects that are also the concern of this other research. In the next section I review other work related to understanding programs and its connection to this thesis.

3.6 On Understanding Programs

This review includes writing understandable programs, but it is mainly a review of work in AI and Software Engineering on understanding written programs, and a discussion of their relevance to this thesis. I differentiate between two issues: areas and techniques. I review areas of research that involve understanding written programs: rational reconstruction, tutoring systems, validation of knowledge-based systems, reverse engineering, and compilation-interpretation-debugging. I also review techniques used to understand written programs: code inspection, program transformation, and automatic program recognition.

3.6.1 Writing Understandable Programs

I have argued for the use of prototyping as a programming methodology in AI research instead of programming from specifications. Still, many of the guidelines and recommendations for obtaining a good style of programming that come from the top-down or layered approach to programming are very useful for writing programs for AI research. [Kernighan & Plauger 74], and [Ledgard 75] give summaries of general guidelines on programming style. The aim of such guidelines is the construction of well-structured and readable (therefore understandable) programs in a systematic way. These guidelines include the organisation of thoughts and planning before writing a program, modularity, and creation of programs from small chunks, localization of any information to be used inside a module or a procedure, specification of input-output of a procedure or module, and readability and documentation of the program text.

Most books on AI programming include these recommendations together with others related to data abstraction, late binding, interpretation (rather than compilation), the use of recursion, debugging, etc. For some comments on programming style for AI programs see [Kreutzer & Mckenzie 91, page 78], and for requirements and features of AI languages that are important for AI programming see [Luger & Stubblefield 89, page 190]. The more these guidelines

are put into practice in the building process of a program, the easier its analysis will be. The Prototype Analysis Method I have developed makes use of modularity to identify functionalities in a program, strong typing for increased precision, and procedural abstraction, and persistence for information hiding. Thus, the more these elements are used for building a program, the easier its subsequent analysis will be.

3.6.2 Rational Reconstruction

Rational reconstruction is an approach to assess the value of published claims about AI programs. It involves the reproduction of the essence of a program's significant behaviour with another program constructed from descriptions of the supposedly important aspects of the original program [Partridge & Wilks 90, page 235]. The rational reconstruction of a program is usually performed by researcher(s) other than the developer(s), and the usual sources of information are published description(s) of the program (usually from the researcher that built it) and sometimes the program itself. Anecdotally, Campbell, [Campbell 90], views rational reconstruction more as an European activity than an American one. The process, he says, usually involves starting from the published information about an American program and trying to reconstruct or reproduce (usually unsuccessfully) the performance of the corresponding program or method. Examples of rational reconstruction are the work of Knapman on analogy-formation as reported in Winston's Ph.D. thesis, [Knapman 78], Bramer and Cendrowska's on MYCIN and expert systems, [Bramer & Cendrowska 84], Ritchie and Hanna's on Lenat's AM system, [Ritchie & Hanna 84], Stokes' on a circumscription algorithm [Stokes 89], and Roberts' on both Wilson's Animat and Holland's CS-1, [Roberts 89], [Roberts 91, chapter 4].

Rational reconstruction is understood as a different process by Bundy [Bundy 90]. In his view, after a program is built and analysed to understand, say, a technique that underlies it, the identified technique is generalized and a *rationally reconstructed* form of it is formulated. In this view, rational reconstruction involves generalization and formulation, not building a new

program. I don't consider this view further here because it does not involve program understanding, as the analysis of the program is done before the rational reconstruction process starts.

Reported work in rational reconstruction does not provide a method to perform the analysis of programs to obtain the essence of their behaviour. One reason for this is that research in the area is still focusing on reconstructing particular systems, not on investigating how the rational reconstruction of a system can be performed in a systematic way. Having good descriptions of the system is vital for the process. Stokes reports on several errors in the original paper describing the algorithm he attempted to reconstruct. Roberts explains that despite the amount of time and effort spent in attempting to get the reconstructed systems to perform as documented, the reconstructions did not match the performance of the originals. He concludes that it remains for the researchers to do the best they can to provide all of the salient information of their system, and that making the code available is useful. One of the conclusions Ritchie and Hanna draw from their analysis of AM is that there is a need to find ways of reporting work in a detailed way (but distinct from implementation details) which will be of direct practical use to successors in the field. I believe that methods like mine can help researchers to obtain, and thus offer, such descriptions of their systems.

3.6.3 Tutoring Systems for Programming

A tutoring system is aimed to help a student (often a novice) to learn a certain subject, e.g., programming. Tutoring systems for programming usually examine a program written by a student programmer, and find and explain any errors in it. In order to examine a program and find errors, a tutoring system must be able to understand programs in an effective way. This usually involves having expectations about how actions can be organised and coded, knowing what is considered correct and incorrect in a program (what are the common sources of error in program writing) and, most importantly, having the specification of the task the program is supposed to accomplish.

Intelligent tutoring systems rely on different techniques for program understanding and error detecting (see [Rich & Waters 86] for a set of papers in the area). Some like [Ruth 86] rely on a parser and a grammar to detect errors. The grammar specifies the correct syntax of algorithms expected in student programs, and it is used to parse student programs. The analyser in this case understands a program by discovering the *intended* algorithm associated with it. This algorithm is deduced from the specification of the task. Others, like Proust [Johnson & Soloway 86] take a program and a non-algorithmic specification of the program requirements, and find the most likely mapping between them. Proust does this by reconstructing the design and implementation steps the programmer goes through in writing the program. To help this process, the system contains a representation of bugs, as well as representing what is correct in a program. An interesting underlying idea in Proust's approach is that bugs are not considered properties of programs, they are considered properties of the relationship between programs and intentions.

An important feature of research in tutoring systems is the recognition of differences between the intentions of a programmer and the resulting program. This is a differentiation I also understand to be important when analysing AI research programs. Despite this common feature, important differences remain between work in programming tutors and this thesis. Programming tutors rely on correct answers or intentions against which the student program is checked. It also relies on a full specification of the task to be performed by the program under analysis. In the analysis of an AI research program there are no correct answers or full specifications of the task, therefore techniques and algorithms developed in this area are not of real use in this thesis.

3.6.4 Validation of Knowledge-Based Systems

The aim of validation in knowledge-based systems is to assure their correct structure and adequate functionality [Lopez *et al* 90] which involves understanding the written program. Depending on whether the main concern

is the structure or the behaviour or the system, validation can be divided into structural verification, functional verification, and evaluation [Lopez *et al* 90].

Structural Verification

In classical software, program verification is concerned with the formal proof of the correctness of a program against a given specification. Rich and Walters, [Rich & Waters 86, section II], describe automatic theorem proving as the ideal way to perform verification, although they recognize the impossibility of the task when applied to complete software systems of commercial size. Instead, current research is oriented towards incremental verification—proving individual properties of a program, and the correctness of small pieces of code. At the same time, the idea of fully-automated systems is being substituted with semi-automated systems that assist the programmer in the development of proofs.

Structural verification of knowledge-based systems (KBS) usually involves a different process to that of classical software verification. One of the reasons is the lack of clear specifications when building KBS [Preece 90], [Lopez *et al* 90]: without a clear specification, no formal proof of a program's correctness can be made against it.

Structural verification in KBS involves determining aspects of a program's structure such as its internal self-consistency and completeness. In rule-based systems, for example, it involves checking consistency and completeness of the rule base by syntactic inspection and manipulation of rules as logical expressions: comparison of individual rules (e.g., two rules that have the same left-hand side and different right-hand sides are contradictory) complete consistency and redundancy checks on the whole knowledge base (checking redundant or subsumed rules) or completeness checks (a given situation that is not covered by any rule) [Preece 90].

This syntactic inspection is a kind of static analysis, and as such is of interest to this thesis. On the other hand, the literature is restricted to knowledge bases, and in particular to rule bases (see [Lopez *et al* 90] for a good review of systems for

structural verification). The method I have developed is aimed at analysing more aspects of a system than just the rule base. In fact, the issue of consistency and completeness of knowledge bases has not been directly addressed in this thesis. Future research could involve investigating work in this area to incorporate this feature in the Prototype Analysis Method.

Functional Verification

Functional verification (sometimes also called 'testing', or simply 'validation') is concerned with checking the behaviour of a program. It involves checking that it does what is supposed to do and also that it does not do what it is not supposed to do. For this, the program is executed against a selected set of test cases to verify that it is adequate for the given requirements and that it does not contain errors. Thus, functional verification is a kind of dynamic analysis.

Contrary to structural verification, functional verification, in Preece's terms, involves the same kind of testing in classical software and in knowledge-based systems (he uses the term validation for both areas). Validation determines if a system satisfactorily performs the real world task for which it was created. The difference in validating KBS and classical software relies in the availability of clear requirements against which to validate the system [Preece 90].

I consider functional verification to be an aspect of dynamic analysis (evaluation is another). As I have explained in section 2.4.1, the investigation of this kind of validation is not the concern of this thesis. However, it is related in that results from a system's dynamic analysis are used during the application of my method. (See also [Lopez *et al* 90] for a review of current efforts at developing a methodology for functional verification.)

Evaluation involves judging or assessing the worth or goodness of a system, and it depends on the evaluator's expectations of the system, [Preece 90]. A way of viewing evaluation is as a further kind of testing to that of functional verification.

An important aspect of the evaluation in AI applications is user acceptance. Preece characterises user acceptance (sometimes also called 'usability') as encapsulating ergonomic and organisational aspects of the system. It can involve testing the system where it is going to be used and by the people that will use it, that is, testing it *in situ*. Evaluating an AI research system usually involves testing the program and analysing the results in terms of a few attributes considered of interest in the experiment. For example, if we build a program to try a new search algorithm, we will need to test the algorithm to see how well it performs, and under which circumstances. Some important examples of evaluations performed as part of experiments to recreate intelligent behaviour include the evaluation of GPS (General Problem Solver) [Newell & Simon 72], MYCIN [Buchanan & Shortliffe 84], and Soar [Laird *et al* 87]. To illustrate the point, I will expand on the case of Soar.

The research aim behind Soar is the development of a general architecture for intelligent behaviour. This means providing the underlying structure that enables a system to perform the full range of cognitive *tasks*, employing the full range of *problem solving methods* and *representations* appropriate for the tasks, and *learning* about all aspects of the tasks and its performance of them [Laird *et al* 87]. Evaluating Soar is very difficult, but also very important to substantiate claims about the architecture and about a system implementing it.

An implementation of such an architecture is presented in [Laird *et al* 87]. The authors admit that the system only realizes part of the capabilities of general intelligence, with significant aspects still missing. Nevertheless they consider that they have obtained enough results to present it as an step towards an architecture for general intelligence. They present a summary of the performance scope of Soar in terms of *tasks* performed, and problem solving methods and

representations used by the system [Laird *et al* 87, page 3]. In [Laird *et al* 86] the authors concentrate on the description and evaluation of two methods—universal subgoaling (a problem solving method) and Chunking (a learning method). In [Rosembloom *et al* 91] they present a preliminary analysis of the Soar architecture as a basis for intelligent behaviour. This analysis is performed in terms of the set of tasks that are natural for Soar, the sources of its power, its scope, and limits.

Another example of evaluation at a different scale is the analysis presented in [Balkany *et al* 91]. In this case, the authors suggest that a Knowledge Level analysis of design tools will allow understanding different tools and their comparison. To sustain this claim they present an initial Knowledge Level analysis of five design tools. Their analysis method consists of identifying a system's problem solving method and its associated mechanisms to attempt to capture the overall behaviour of the system.

Evaluation (like functional verification) is not the concern of this thesis. Still, it is important to have an awareness of previous efforts on evaluating AI research programs because it is complementary to the static analysis of programs.

3.6.5 Reverse Engineering

The term 'reverse engineering' was originally coined in the area of analysis of hardware. Hardware reverse engineering is used to improve a product or to analyse products from competitors, and the objective is to duplicate a given system. It involves examining hardware systems, and developing a set of specifications from them. The result of applying some of these concepts to software is what is understood as software reverse engineering (or simply reverse engineering, as I will call it from now on).

In software, reverse engineering is defined as the process of analysing a system for two purposes: to identify the system's components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction. (See [Chikofsky & Cross 90] for a unifying

terminological taxonomy that differentiates reverse engineering from other related areas like forward engineering, re-structuring, and re-engineering.)

Reverse engineering is applied mainly in the process of maintenance of a system. The objective is to obtain a better understanding of a program to aid its maintenance. It usually involves extracting design artefacts and building or synthesizing abstractions that are less implementation dependent. It is the part of the maintenance process that helps understand a system at a certain level of abstraction, so that appropriate changes can be made successfully. It is important to note that reverse engineering does *not* involve changing or replicating the original system. It is just a process of *examination* of a given system. Changes to the system are performed by other parts of the maintenance process such as re-engineering or re-structuring [Chikofsky & Cross 90]. Usually the reverse engineering task is performed by somebody other than the original developer. The systems examined are mainly classical commercial systems that may be rather old (even 10 or 15 years) and documentation may be scarce or even non-existent.

To help this complex process, a lot of work in the area is concerned with the development of tools that can make reverse engineering easier. Exactly what kind of information is obtained from the process and the use of these tools depends on the objective. The result can be pictures, diagrams, metrics, logic, reports, documentation, etc. For example, re-documentation is a subarea of reverse engineering that creates or revises semantically equivalent representations within the same abstract level, and the result can be pretty printers of code, diagrams of the control flow or code structure, etc. By contrast design recovery is another subarea that uses a combination of code, any available design documentation, and programming, problem and domain knowledge to recreate all the relevant information about a system.

Artificial Intelligence techniques are used to build some of these tools [Harandi & Ning 90], [Rich & Wills 90], but I found no reports of the application of research in reverse engineering to AI application systems or AI research systems. Still, tools from reverse engineering can be very useful in AI research. In rational reconstruction, for example, they could be used by a researcher

attempting to rationally reconstruct a system to understand better the original system.

The work in this thesis bears a strong relation to reverse engineering. They are related in that both are concerned with understanding a system by obtaining more abstract levels of description. However, strong differences also exist. The aim of the understanding obtained from the process is different—understanding a program's causal structure is different from understanding particular parts of a program to support its maintenance. The Prototype Analysis Method incorporates Software Engineering abstraction constructs to help the analysis process. Reported work on reverse engineering is not concerned with the use of these abstraction constructs to help the process of reverse engineering. Other differences relate to who performs the inspection of the code—the method I have developed is used by the researcher who built the system being analysed, and not by somebody else as is typically the case of reverse engineering. Also, the Prototype Analysis Method involves changing or transforming the original program on the way to obtaining a higher level description of it, whereas reverse engineering only involves examining a program, not changing it.

I believe that tools developed in reverse engineering could be useful to help performing or automating parts of my method (which are done by hand at the moment). At the same time the abstraction constructs used in my method could be used in reverse engineering to help in managing the complexity of the task and in hiding information as they do in the Prototype Analysis Method.

3.6.6 Compilation, Interpretation, and Debugging

To finish this review of areas that involve program understanding, I will briefly mention compilation, interpretation and debugging. A compiler, interpreter or debugger needs to have some understanding of the program to perform its task. Compilers and interpreters perform static analysis on the source code to translate it. Debuggers perform dynamic analysis of the program to find the source of an error. Unlike in this thesis, understanding a program in these contexts usually means checking either the code or the behaviour of a program against a set

of usually well defined rules. However, they are worth mentioning as further examples where static and dynamic analysis of programs are performed.

3.6.7 Techniques for Program Understanding

Having looked at areas of research that involve understanding of programs, here I concentrate on techniques used to help the understanding of programs: code inspection, program transformation, and automatic program recognition.

Code Inspection

Code inspection is the basic activity where actual program code is looked at to extract some kind of information from it. It is an integral part of the static analysis of a program, which is the reason for mentioning it here. However, I have not found references in the literature that address the issue of code inspection as such. I believe the reason is that looking at code is such a basic activity that it would be difficult to address it other than as part of another activity, a technique like automatic program recognition, or an area like tutoring systems, for example.

Code inspection is common to the areas reviewed and my method. All of them perform some kind of code inspection in one way or another—manual or automatic—and it is performed by the builder or others. All of them obtain some kind of information—structural, control flow, metrics, design decisions etc. In the Prototype Analysis Method inspection of code is performed manually by the same researcher that built the program, and it is aimed at identifying particular instances of data structures used in the program, and operations performed on them.

Program Transformations

Program transformation is the replacement of a part of a program with a new part. The aim of a transformation is to improve the old part in some sense, usually the new part is more efficient, or more specific. At the same time, the transformation has to preserve the functionality of the original part. That is, the

new part has to compute the same thing as the old part. It may compute it in a different way but the result has to be the same (it may change the how but not the what).

There are mainly two kinds of program transformations: vertical and lateral. *Vertical* transformations transform a program to a more specific (or implementation dependent, thus, less abstract) form. When a vertical transformation is applied an expression described at a high-level of abstraction (using specification-like constructs) is defined in terms of an expression at a lower level of abstraction (using implementation or language dependent constructs) e.g., defining how to sort a list using a loop. This is also sometimes called transformational implementation. *Lateral* transformations specify an equivalence between two expressions at a similar level of abstraction, e.g., change the order of some instructions if they are commutative to make the program more efficient. Lateral transformations are used mainly to convert a clear but inefficient program into an efficient one. It can also be used to set things up properly so that vertical transformations can be applied. (See the section on transformational approaches in [Rich & Waters 86] for a set of papers in the area.)

Program transformation is a very active area of research in automatic programming. Its main concern is the development of systems that perform transformations automatically (either completely automatic or guided by the user). For example Darlington, [Darlington 86], reports on a system that performs lateral transformations in a program to make it more efficient. For that, the system applies a very small number of powerful general-purpose lateral transformations to merge information from different parts of the input program and then redistribute them into a more efficient modularization. The result is a more efficient but a much less clear program. Transformational implementation is an approach that makes use of both vertical and lateral transformations [Rich & Waters 86, Introduction] to transform clear programs written using high-level constructs into an efficient form. This is done by replacing the high-level constructs for lower level ones by iteratively applying lateral and vertical

transformations. In this case large libraries of special-purpose transformations are used.

The Prototype Analysis Method, and in particular the transformation process that is part of it, involves what could be understood as a reverse vertical transformation. In this case, language dependent constructs are transformed to a less implementation dependent form. A program transformation has three parts: patterns to be matched against parts of a program, a set of applicability conditions for the matching, and an action that creates a new part to replace the part matched by the pattern. Usually the patterns are implementation independent whereas in this case they would need to be implementation dependent, and the replaced parts are implementation dependent which in this case would need to be the contrary. Thus, automatic tools developed as part of research on vertical transformations are not straightforwardly applicable to this work.

The transformation process is currently manual. I believe that the application of the method, and in particular the complexity management involved in applying the method to a large AI program, would improve if transformations were performed automatically. I will point to how this can be improved in section 7.5.4.

Automatic Program Recognition

Program Recognition is a technique used to statically understand programs based on the recognition of clichés in a program [Wills 90]. Clichés are commonly used programming structures, usually data structures (i.e., sorted lists, hash tables, etc.) and algorithms (i.e., list enumerations, binary searches, etc.). The idea behind cliché recognition for program understanding comes from observation of how experienced programmers understand programs. Programmers rely much more on their accumulated programming experience when analysing programs than on reasoning from first principles. An experienced programmer can recognize parts of a program by identifying familiar computational structures in source code, and he or she knows how these structures typically implement higher level abstractions.

Wills, [Wills 90], presents an attempt to automate program recognition, a system called the Recognizer. The practical motivation for automatic program recognition is to facilitate many of the tasks in Software Engineering that require understanding programs—maintenance, documentation, enhancement, optimization, and debugging. Automatic program recognition attempts to facilitate these tasks by reconstructing program design information. It is also considered an interesting theoretical task for Artificial Intelligence; it allows the study of how programming knowledge and experience can be represented and used.

Wills' system—the Recognizer—uses a library of clichés to find all the occurrences of the clichés in a given program, and builds a hierarchical description of the program in terms of the clichés found and the relationships between them. The approach taken is based on two elements: the representation of a program and the recognition technique. The program is represented in a language independent, graphical manner, based on the Plan Calculus [Rich 81]. In the Plan Calculus a program is represented as an annotated directed graph called a *plan.* The searching and recognition of clichés in the program is made using an exhaustive, partial recognition technique that treats program recognition as a graph parsing task. [Rich & Wills 90] give an account of the limitations of the system and the extensions needed to be of practical use in maintenance.

An interesting point about this system is that it is part of the long running Programmer's Apprentice project that attempts to provide intelligent assistance in various phases of the programming task [Rich & Waters 90]. One of the goals of the Programmer's Apprentice project is to explore how to use the Recognizer to help the task of knowledge acquisition. In particular, they want to investigate how the system might automatically learn new clichés [Rich & Wills 90]. Other recognition systems use, as well as the program and clichés, other information like specifications [Lukey 80], a set of goals [Johnson 86], or a model program that performs the same task [Murray 85].

Automatic program recognition is not a technique used in my method. Libraries with clichés conventionally used in experimental AI programs are not

available. I have discussed the issue of identifying patterns in experimental AI programs, and why this thesis does not attempt to investigate it in section 3.3.2.

However, as part of the transformations, user-defined types and operations are first identified in the program, and then substituted. This process of substitution (which also involves recognition of the previously identified elements) is currently done manually. There was no attempt to investigate its automation during the investigations in this thesis because it was not clear what constituted the process of substitution, that is, how the substitution was to be done (either manually or automatically). It was necessary to understand first what was involved in performing it before any consideration could be given to its automation. After the investigations, I believe the substitution step could be automated (I will discuss how in section 7.5.4) and techniques from automatic program recognition could be used in this part of the method in the future.

3.7 Summary

In this chapter I have discussed the nature of the AI programs whose analysis is the concern of this thesis and the approach taken for the investigations. I have presented my thesis and a sketch of the Prototype Analysis Method that I have developed during its investigation. Other work on understanding programs has also been discussed in relation to this thesis.

Chapter 4

Program Analysis and Abstraction Constructs

Software Engineering abstraction constructs are investigated in this thesis to help the static analysis of experimental AI programs. In chapter 3 I argued that the analysis of programs is a neglected aspect of AI research, and I presented the approach taken in this thesis to investigate program analysis. In this chapter, I will discuss some important aspects of the analysis—information hiding, increased precision, and complexity management. I will present research in Software Engineering on abstraction to support these aspects, and how they can be achieved computationally. Finally, I will review abstraction constructs and identify those investigated, and explain why the Napier88 language was chosen for the experiments.

4.1 Aspects of Program Analysis

Abstraction can be characterised as the mapping of one representation of a problem, the *ground* representation, onto a new, less detailed, representation, the *abstract* representation [Bundy *et al* 90].

Abstraction is a concept, and it is of use wherever understanding and conceptualization are required. In science, for example, when we form theories to describe classes of phenomena, the significant quantitative and qualitative

features of the class are abstracted out from the details that characterise its individual members. This loss of detail is compensated for by the descriptive, explanatory, and predictive power of a valid theory.

From a practical point of view, abstraction is also considered the best known method for controlling complexity in building systems because it allows the details of a particular level to be ignored when viewing the system from a higher level [Morrison *et al* 88]. Conceptually, an abstract representation is easier to handle because it contains fewer implementational details.

Next, I discuss three kinds of aspects of analysing AI programs and why they are important in achieving higher degrees of abstraction—information hiding, increased precision and complexity management.

4.1.1 Information Hiding

Information hiding is the process of separating and hiding the details of a specific implementation (inside some body which offers an interface to the user) from the specification of its functionality. The user is thus presented with the specification (via the interface) of the functionality of the implementation hidden in some body. That is, information hiding abstracts the *what* from the *how*.

Achieving higher degrees of abstraction is the aim of the transformations performed on the program: a program is transformed from an implementational and language dependent form to a more abstract form which hides those details. Information hiding is important to support this process. It abstracts the specification of user-defined types and their operations from their implementation, and how they are used in the program. It helps in understanding whether something refers to a design decision or an implementation decision. For example, a certain operation might be designed for some general purpose (e.g., an identification or equality operation). The application of the operation to particular cases occurs during its use (e.g., identification of strings or integers). However, it may be that the application of the operation is correct (e.g., an operation that identifies strings) not because of the way the operation is used

(general identification applied to strings) but because of the way it is implemented (directly as identification of strings). Using information hiding it is easy to see if an operation is defined and implemented for general identification, and used for the particular case of strings, or if it is implemented and used only for strings. Thus, it helps to understand how much is due to the way a program is designed and how its components are used, and how much is due to its implementation.

Another important kind of information hiding involves hiding, or at least separating, different functionalities in a program. This supports the analysis by allowing us to concentrate on the interpretations and transformations of specific (and separable) parts of the program, instead of analysing the whole program at once. I call these parts modules, and this is investigated as part of the transformation method developed in the next chapter.

4.1.2 Increased Precision

Understanding the structures in a program and their representational role involves a process of interpretation. This interpretation involves defining the role that particular user-defined types and operations have in the program and using them accordingly. Obtaining reliable interpretations of user-defined types and operations is not necessarily a straightforward process, and it is important to support the researcher when performing these interpretations.

Increased precision is the process of imposing constraints on the definition of objects and on the interaction between objects in a program. By providing increased precision, objects are prevented from having inconsistent interaction with other objects. Increased precision supports a researcher when performing the interpretations by ensuring the reliability of the definition and use of data structures and operations identified in a program, and thus, their interpretation.

4.1.3 Complexity Management

Experimental AI programs are usually complex, and their analysis is thus likely to be a complex and difficult task. There are two sources of complexity which I

call structural and managerial. Structural complexity refers to the difficulty of the task in hand, the intended analysis: identifying user-defined types and their operations, and performing the program transformations.

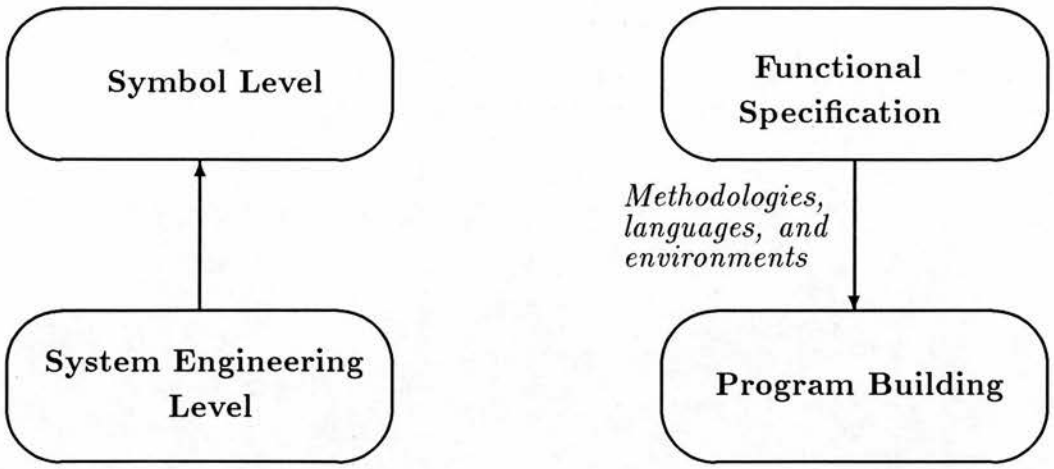
The identified user-defined types and their operations need to be managed, changes resulting from new or modified interpretations and corrections of transformation errors need to be performed in a reliable and organised way. This is what I call managerial complexity: management tasks which provide a further source of complexity to the analysis of a large program.

Abstracting the analysis task from the complexity of its management is what I call complexity management. Complexity management is an important aspect of the analysis. It allows the researcher to concentrate on the interpretation and transformation task, rather than being distracted (and possibly overwhelmed) by details of their management.

4.2 Software Engineering

Understanding the computational concepts required to engender particular kinds of behaviour in computer programs is one of the concerns of Software Engineering research. I take Software Engineering research to be the discipline concerned with the formalization of software production. The fundamental aim is to develop formal concepts and general principles, mechanisms, and techniques to understand computational systems and the relations among system components, as well as the mathematical modeling techniques associated with these systems.

The Software Engineering community is concerned with the need to construct programs that are correct with respect to a given specification. Implementing a system that accomplishes a given specification is especially difficult when building large complex systems. Research in Software Engineering (and in particular in programming languages) is concerned with controlling the complexity of a system to allow the developer to concentrate on the specified functionality rather than on the details of construction which are irrelevant to the specification. Providing



(a) AI Research (b) Software Engineering

Figure 4–1: Relationship between AI research and Software Engineering.

more powerful and more coherently combined computational constructs (e.g., strong data typing, polymorphism, and persistence) that allow higher degrees of abstraction is the concern of Software Engineering.

These concepts and paradigms are realised by computational constructs that are usually implemented by embedding them in a programming language. Modern programming languages which embody such concepts enable program builders to work at more abstract levels, thus freeing them from details that are not of specific relevance to the functionality of the particular program. If such constructs can help in going from a Functional Specification to a program (see figure 4–1(b)) then perhaps they can also help in going from a program to an abstract Symbol Level description of it (see figure 4–1(a)). In other words, if going from a functional specification to a correct program is similar (but in the reverse direction) to going from a program to an abstract Symbol Level description, then the computational concepts and techniques designed to help the former might also help the latter.

4.3 Software Engineering and Abstractions

Here I discuss developments in Software Engineering that provide the three aspects of the analysis described in 4.1.

4.3.1 Achieving Information Hiding

Investigating concepts and computational constructs to achieve information hiding is an active area of research in Software Engineering. Various degrees of information hiding can be achieved depending on the facilities used. *Encapsulation* is the term used to describe those features of a programming language which support information hiding [Thomas *et al* 88]. In the procedural abstraction paradigm where the emphasis is on the operations (data are viewed as the elements supplied to operations) encapsulation is provided with procedures or functions. In this case, the implementation details of a certain operation are hidden inside the body of the procedure or function. However, the implementation details are not very well hidden because they can be seen and usually accessed just by looking at the code. Also, depending on the requirements of the language, the interface provided by the procedures or functions may not be very precise. For example, some languages don't require the declaration of parameters and/or results of a procedure or function. Most languages provide more information hiding by extending the procedural abstraction to that of libraries.

In the data abstraction paradigm the emphasis changes from operations to data (thus, operations are applied to data). In this paradigm encapsulation is provided by abstract data types, which is extended with object-orientation in some languages [Pascoe 86]. As we will see later, packages (in Ada) and modules (in Modula-2) are examples of facilities found in programming languages for data abstraction, and thus for real encapsulation in a programming language.

Another kind of data abstraction that also provides information hiding is allowing a type structure or operation to be defined abstractly from the particular data type instances it can contain. This is usually called polymorphism.

A further information hiding mechanism that a programming language can offer is persistence. Persistence is a property of all data that determines how long it exists. Usually, persistent data is understood to mean long-lasting data. In all the previous cases, the encapsulated data and operations are kept in files that need to be loaded, compiled or linked with an application program, for the application program to use them during execution. With persistence the implementation details are hidden away, and the application program needs only use the interface between the persistent store and the program to use the hidden elements. Furthermore, the persistent character of these operations means that no further loading or compilation or linking is necessary each time an application program is run.

Thus, the mechanisms investigated for information hiding are: the data abstraction paradigm to encapsulate data structures with their operations, polymorphism for higher data abstraction, and persistence to achieve a higher and long-lasting degree of information hiding. Procedural abstraction is also used to encapsulate operations identified during the analysis.

4.3.2 Achieving Increased Precision

Research in programming language semantics is concerned with putting programming on a firm foundation with practical semantic theories. It seeks to have programs and the objects represented therein with unambiguous values in some semantic domain. Types are seen as providing a way of representing such meanings [Danforth & Tomlinson 88], and type checking systems are developed to ensure the reliability of their representations.

The use of types (together with a type checking system) is investigated because it is a good mechanism to provide increased precision: the type system supports the researcher in defining the data structures and operations identified during the analysis, and it ensures their correct use. The researcher only needs to concentrate on the identification and transformation task leaving to the type checking system the task of verifying their correct definition and use.

4.3.3 Achieving Complexity Management

Mechanisms that help achieving complexity management are necessary. At the same time, it would be good if these mechanisms had firm foundations, based on some theory or some well-defined properties. An *adhoc* mechanism may not be of much help since it is difficult to make sure that the transformations performed during the analysis don't change the behaviour of the original program.

As we have seen in section 4.2, Software Engineering is concerned with developing constructs based on well understood principles. They are aimed to support the construction of programs that are correct with respect to a given specification, and in such a way that they also support managing the complexity of the construction task. This is usually achieved in two ways:

- Providing constructs that enable programmers to work at more abstract levels and in a way that makes them easy to use.

- Providing a uniform environment where all the constructs are available in a coherent way.

I have investigated those constructs that support complexity management during the analysis (as well as information hiding and increased precision). Thus, the specific abstraction constructs investigated (i.e., the specific kind of abstract data typing, polymorphism, persistence, and types) have characteristics that make them easy to use during the analysis, and are available in a uniform environment. As a result, they should support the management of the identified user-defined types and operations, and the performing of transformations. Also, they should support performing changes to the identified elements, as a result of new or modified interpretations, in a reliable and organised way.

4.4 Specific Abstraction Constructs

In this section, I discuss various specific kinds of persistence, types, abstract data typing, and polymorphism (using a historical perspective). This discussion shows the characteristics that make the the specific abstraction constructs chosen for the investigations interesting for the analysis (i.e., specific kind of persistence, specific typing, abstract data types, and persistence) and how they are more useful than other kinds.

4.4.1 Persistence

Persistence can be viewed as a property of all data, and it determines how long data exists for. In programming terms existence can be equated with data being potentially accessible by a programmer or a program operation. The terms short term data and long term data are used to denote the kind of existence that data have. Short term data (also called transient data) are data that exist while a program is running (when using programming environments provided by languages like Pascal [Wirth 71]) or during a session (when the programming environment is that of Lisp, [McCarthy 62], or Prolog, [Clocksin & Mellish 81], for example). Not all short term data have the same persistence. Data local to a function persist as long as the function is active, whereas data global to a program persist while the program is active. Thus, global data are more persistent than local data. Long term data (also called persistent data) usually refers to data that outlive the running of a program or a session, and for that, files or databases are used. Thus, persistence is a property that all data have to some degree.

Traditionally, research in programming languages (such as Pascal, Lisp, Prolog) has been mainly concerned with the definition and manipulation of short term data, and research in databases with the definition, and manipulation of long term data, i.e., with the development of data models like CODASYL, relational, entity-relationship, and functional [Hughes 88]. Research on persistent programming can be viewed as a middle ground filling the gap between short and

long term data. In other words, it is concerned with how to create interfaces between programming languages and persistent data.

Here I give a brief review of the different approaches to persistent programming, and their historical development (see [Atkinson *et al* 84] for a more complete review on persistent programming). The most basic kind of persistence offered by most programming languages is the sequential file. Operations on files involve mainly input/output operations which are provided in a library of separately defined procedures that are called within a program. Having a library of procedures to provide database facilities in a language is known as the *call interface method*, and it was the earliest attempt to have a systematic mechanism for persistent data. In this approach the programmer has the flexibility to define, build, and manage the data structures corresponding to the persistent data. The main problems with this approach are that the system does not provide consistent typing, nor consistent rules for introducing and using names. The programmer has to control the consistency between the structure of the persistent data, and its use in the program, that is, the programmer is responsible for the matching between the physical storage of data and their logical (more abstract) use in the program.

Embedded DML (Database Management Language) is an approach that attempts to solve some of the previous problems. The DML keeps a description of the persistent data, and the programmer mixes in the program statements from the programming language and the DML. An example of this approach is COBOL + CODASYL [Knowles & Bell 84]. The resulting program is checked by a preprocessor, and then compiled (in some systems, the preprocessor is part of the language compiler). In this way the DML takes over the responsibility of the matching between the physical and logical description of the persistent data, and some checking is performed. The disadvantage of this approach is that the programmer has to worry about mixing the two kinds of statements in the program, that is, he or she has to work with two programming languages at the same time (this problem is known as impedance mismatch).

Further research in the area can be characterised as research on *database*

programming languages whose aim is to develop languages that manage short and long term data in a coherent way. Following [Atkinson *et al* 84], I describe two of the latest approaches in database programming languages—*integrated programming languages* and *persistent programming languages.*

The aim of integrated programming languages is the integration of existing programming languages and data models to derive a coherent whole. In this approach programming languages are extended with database facilities. For example, Pascal/R, [Schmidt 77], extends Pascal to include facilities of the relational data model. It incorporates the data structure `relation` to the set of data structures in Pascal, together with some facilities to manipulate it. ADAPLEX, [Smith *et al* 81], extends the programming language Ada, [Ichbiah *et al* 79], to include facilities of the functional data model DAPLEX [Shipman 81]. An advantage of this approach is that both short and long term data are treated in the same way. However, not all data have the same properties: only the data structures in the language that correspond to the data model are persistent (e.g., `relation` in Pascal/R). Using this kind of language means that the programmer needs to maintain two conceptual models (at the same time) while writing a program—the database model and the program model.

The persistent programming paradigm aims to treat long term data in the same manner as short term data. There are two important concepts that make this paradigm a step forward with respect to the other kinds of persistence. One is that all data structures that can be manipulated in the programming language have the same right to persist. That is, there are no special data structures for the database incorporated into the language (as in the case of integrated programming languages). This is called *orthogonal persistence.* In this paradigm there aren't two different models—program model and database model. The program model is the only model the programmer needs to worry about. The paradigm provides the program builder with a uniform model of data over time.

[Atkinson *et al* 83] identify the principles of *persistence independence* and *persistent data type completeness* as two crucial rules for persistent programming. Persistence independence means that the persistence of a data object is

independent of how the program manipulates it, and conversely, a program is expressed independently of the persistence of the data it manipulates. Data type completeness means that all data types in a language have the same 'civil rights'. In line with this principle, persistent data type completeness gives the full range of persistence to all data objects in the program. That is, not only all data structures are given the same right to persist, but also functions or procedures. In the classical data models (i.e., CODASYL, and relational) only certain data items can be stored in the database (e.g., attributes of a relation in a relational data model can only store atomic values). By allowing procedures the same right to persist as data, it is possible to keep program and data bound together. This allows incremental and localised building and persistence of data structures and their operations, thus, providing an powerful and simple mechanism for sophisticated information hiding, as well as complexity management. For these reasons, persistent programming was chosen as the kind of persistence to be investigated.

Examples of persistent programming languages are Napier88 [Dearle *et al* 90], Amber [Cardelli 84] (a persistent derivation of ML [Milner 83]) GNU E [Richardson & Carey 89](which extends C++ with the notion of persistent data) various persistent Prologs—Perlog [Moffat & Gray 88] and the persistent Prolog described in [Colomb 88]—and the persistent Object-Oriented Lisp LISPO2 [Barbedette 90].

Perlog is a Prolog with persistence and modules. The motivation for the development of Perlog was that the database provided by Prolog has the drawback of being a single, and flat, clause base. In Perlog the clause base is partitioned into modules that can persist between sessions. LISPO2 is a persistent Object-Oriented Lisp, where persistence is introduced as an extension of the lifetime of Lisp data. This is done by defining some persistent roots (database variables and class extensions) that are added to the other Lisp roots (the symbol table and the stack). Thus, everything in the closure of class extensions and database variables is persistent in LISPO2. Some derivatives of Lisp like INTERLISP, [Teitelman 75], provide a kind of orthogonal persistent data. It consists of a

mechanism to save and restore the current workspace. This mechanism just copies the whole workspace out to a file. The drawbacks of this method are more extensively described in [Cockshott *et al* 83] but some of them include the fact that the workspace must always be loaded onto the same location in memory, otherwise pointers held in the workspace become invalid. It is impossible to store more data than the amount that can be loaded onto the workspace at any one time. Also, it does not allow incremental and local storage of the workspace. Even when small changes have been performed to the workspace, the whole workspace has to be saved at the end of a session.

Napier88 offers orthogonal persistence, and it aims to satisfy the principles of persistence independence, and persistent data type completeness.

4.4.2 Types

Types can be characterised as sets of objects with uniform behaviour. If we consider the universe of all possible computable values, a type is a subset of that universe. However, not all subsets are legal types. The subsets of the universe that are legal obey some technical properties, and they are called *ideals*. Thus, all types found in programming languages are ideals [Cardelli & Wegner 85].

Considering types as sets of objects with uniform behaviour helps to understand the need for typed languages. For example, let's consider the case of pure Lisp. Pure Lisp [McCarthy 62] is an untyped language in the sense that it only has one type, the S-expression; i.e., its universe is constituted only of S-expressions. Program and data are not distinguished, and everything is an S-expression of some kind.

However, we are likely to need to organise a universe in different ways to distinguish the usage and behaviour of different sets of objects, and thus achieve different purposes. Designers of early Fortran (a language designed mainly for numerical computation) distinguished between integers and floating-point numbers: integers were good for iteration and array indexing, and floating-point numbers were necessary to represent the values to be computed. Thus, types can

be viewed as a consequence of the need to organise an untyped universe. The first important language to have an explicit notion of type and type checking was Algol 60, the first language of the Algol family. Algol 68 [Van Wijngaarden *et al* 69], Pascal [Wirth 71], ps-Algol [Carrick *et al* 87], and Napier88 [Morrison *et al* 89] are also members of the Algol family.

A type system must check that the types defined by the system are used correctly, and that their technical properties are preserved. This is usually called *type checking*. By performing type checking, typed programming languages impose constraints on object interaction in a program. These constraints prevent objects from having inconsistent interaction with other objects. Not all typed languages offer a uniform model for their type system. Strongly typed languages are languages in which all expressions are type consistent [Cardelli & Wegner 85]. That is, the type system checks that all the expressions in a program are free of type errors. Some languages like Pascal do not provide a complete or strong type system in that they don't require the type of certain values to be specified (nor do they infer it themselves). Other languages like ML, [Milner 83], do provide a strong type system that does not require explicit specification of types: ML provides a type inference system which infers the type of an element if it has not been explicitly stated by the programmer.

A type system describes not only the types of elements that can be represented but also the relationships between types, i.e., type equivalence, and type inclusion. Describing relations amongst types involves performing computations on them to determine whether they satisfy the desired relationship. The most basic relation among type expressions is *type equivalence*. Type equivalence determines when two type expressions are of the same type. Ada, provides a weak sort of type equivalence, called *name equivalence*: each time a new name for a type is introduced, a new type is created. A stronger notion of type equivalence, structural equivalence, is provided by languages like Algol 68 and Napier88. In structural equivalence, for two types to be equivalent they must have the same structure regardless of their names. Thus, a type expression can be denoted by several names, but new names do not create new types.

Strong type systems are not usually used in AI languages, where weakly type systems are preferred. An important question is whether a strong type system can be used to build AI programs. Expressive power and flexibility are important aspects when building AI programs. AI languages usually provide them by not distinguishing between programs and data and providing dynamic binding. Traditionally, strong typed systems (usually understood as static) have been considered inappropriate for AI programming because of their lack of flexibility. Traditional statically typed systems exclude techniques that are incompatible with early binding of objects with types (e.g., generic procedures like sorting that are applicable to a range of types—which type it is applied to being decided at run time). However, we will see that some modern strong type systems incorporate two important characteristics that provide the expressive power and flexibility necessary for AI programming—data type completeness and a mixture of static and dynamic checking.

An important question about type checking is when it is performed. Depending on the facilities provided by a type system there are three categories—statically typed languages, dynamically typed languages, and languages that offer a mixture of static and dynamic typing. Statically typed languages are those that check the type of the expressions at compile time before the program is run (static checking). Examples of statically typed languages are Pascal and Algol 68. In dynamic typing (or run-time checking) the correctness of the type of the expressions is checked while the program is running. A classical example of dynamic typing is Common Lisp. Static checking is safer than dynamic checking because it allows us to say if a program is type correct without needing to run it, that is, by statically looking at the code. Type errors are usually much easier to find during compilation, and a statically checked program is usually more efficient. On the other hand forcing all variables and expressions to be statically bound restricts the flexibility and expressive power of a system. Flexibility is important if we have an evolving or changing system, as we typically do in experimental AI programs. If the system is statically checked any changes to it require the whole system to be recompiled to accommodate the changes. In this case dynamic checking is much more convenient. [Cardelli & Wegner 85] recommend having

strongly typed systems that adopt static checking whenever possible, but that introduce dynamic checking when necessary.

An important principle for greater expressive power and flexible programming is data type completeness. Data type completeness means that all data types in a language have the same 'civil rights', i.e., all data types (including procedures and functions) are first class objects, and thus can be used in any combination in the language. Not all type systems achieve data type completeness (e.g., the type system in Pascal does not allow procedures or functions to be returned as results of other functions). However, this kind of flexibility is important for AI programming (and very usual in languages like Lisp). Thus, it is important that a strong type system incorporates this principle for its use in AI programming.

A type system has to provide good increased precision, and a strong type system is a good candidate—it ensures that all expressions are type consistent. It also has to have enough expressive power and flexibility to support the analysis of AI programs. This kind of strong typing is the approach taken by the designers of Napier88 which employs eager type checking, that is, types are checked as early as possible (sometimes statically and sometimes dynamically) and it incorporates the principle of data type completeness.

4.4.3 Abstract Data Types

An abstract data type (**ADT**) encapsulates information that provides a representation of a complex data object and operations that implement the manipulations of which the object is capable. The internal representation of the object represented by an **ADT** is completely hidden from its users, as well as the implementation of its operations. Only the specification of the operations are made available to the user. The abstract data type paradigm also provides flexibility when performing modifications to an **ADT**. If the implementation is modified any code that uses the **ADT** should continue to work unchanged, as long as the external interface remains the same, [Danforth & Tomlinson 88]. In this way **ADT**s provide modularity and information hiding, together with a good interface for the external user of the **ADT**.

Data abstraction can be implemented in many ways. A language may not provide **ADT** facilities, and a programmer could still write a program in an abstract data type style. A language with procedure call mechanisms could be used to obtain a similar abstraction. However, this would involve the programmer developing and maintaining the style and protocols necessary to obtain the data abstraction while developing the program to solve the task in hand. Many languages have attempted to incorporate **ADT** facilities to allow programmers to concentrate only on the task in hand, and most importantly, to provide a principled way of defining and implementing the data abstraction. Such languages include Ada [Ichbiah *et al* 79], and Modula-2 [Wirth 83]. Ada's `package` and Modula-2's `module` are mechanisms to support data abstraction.

Data abstraction, as a way of organising data with associated operations (as opposed to the procedural abstraction) was one of the primary objectives of the design of Ada's `package`. Packages in Ada enable the programmer to group together a set of related routines in a way that hides their implementation. A package has two parts—specification and body—which are kept physically apart. The specification part (which is the only part that is available to the user) contains the type definitions and the names of the routines which implement operations of an abstract data type. The body contains the implementation of the operations. Data abstraction is not completely achieved in Ada because certain types and routines which are part of the implementation may have to appear in the specification part. Ada solves this problem by having a `private` construct that allows the programmer to declare something to be local to the **ADT** in the specification part. In this way, private implementation details won't be accessible to the user. However, they will still be visible (although they are private they are still in the specification part which is available to the user).

Modula-2 is a direct descendant of Pascal extended to include modules. It is the first widely used language to use modularization as a major structuring principle. A `module` contains an interface part that specifies the types and operations available in the module, and the rest of the module (which is kept within a block structure) contains the implementation. Intermodular relations

are possible by means of importing and exporting components of modules. For information hiding it is also possible to split a module in two—definition and implementation—which are similar to the specification and body parts of packages. All objects defined in the definition module are available for export. Unlike Ada, types can be named in the definition module but defined in the implementation module by means of *opaque* export. In this way data abstraction is fully achieved in Modula-2.

Type checking in the previous languages, i.e., Ada and Modula-2, has not been based on any foundational theory giving meaning to **ADT**s. Developing such theories is one of the concerns of research in programming language semantics. Types provide a way of representing meanings, and several type theories have been developed for **ADT**s. They can be classified into algebraic (first-order) approaches and higher order approaches. Algebraic approaches include work on applying algebraic ideas to develop a connection between theory and practice [Burstall & Goguen 82]. OBJ2, [Futatsugi *et al* 85], is an example of a language based on the algebraic approach.

Higher order approaches include work on existential types to express or support **ADT**s. Languages like SOL [Mitchell & Plotkin 85] and Napier88 [Dearle *et al* 90] use the existential types of Mitchell and Plotkin [Mitchell & Plotkin 85]. Other languages like Fun, [Cardelli & Wegner 85], are based on Cardelli and Wegner's existential types [Cardelli & Wegner 85]. Other higher order approaches involve the use of dependent types, with DL as a representative language [MacQueen 86], based on the formulas of constructive logic [Martin-Löf 82]. These approaches take the view that values must be explained in terms of types. Thus, type checking is performed to verify that denoted values are members of, and act according to the properties of, their claimed type. A different higher order approach is taken in languages such as Russel [Demers & Donahue 79], and Poly [Matthews 88], where values themselves do not have types. They are interpreted by the types' operations. Type checking then is the act of verifying that values will not be misinterpreted by operations in

which they are used. (See [Danforth & Tomlinson 88] for a more extensive review of type checking approaches for ADTs).

A kind of abstract data type that incorporates a type theory is an obvious choice for the investigation of ADTs for the analysis. It provides information hiding, and also increased precision—the type checking verifies that the semantics of the ADTs defined are correct. However, which specific type theory approach to choose was not clear, as they all seem to offer good type checking. It is for this reason that the particular approach to be investigated was not decided on the basis of the previous discussion. It was to be decided depending on which approach was available as part of a environment. That is, one that also incorporates, in a coherent way, the other abstraction constructs chosen for the investigations.

4.4.4 Polymorphism

I have defined types as sets of objects with uniform behaviour. However, uniform behaviour does not imply single behaviour. A type can constrain objects of a set to have just one behaviour (type) or more. A type system that constrains objects to have just one type is a *monomorphic* type system, whereas one that allows objects to have more than one type is a *polymorphic* type system.

Originally type systems were monomorphic, and conventional monomorphic languages like Pascal constrain its elements (procedures, variables, values, etc.) to have a unique type. However, strict monomorphism is very restrictive in the expressive power it allows (i.e., variables can't have different behaviour in different contexts) and most monomorphic languages offer a kind of relaxed monomorphism by allowing exceptions like overloading, coercion, value sharing, and subtyping [Cardelli & Wegner 85]. *Overloading* allows the same function name to denote different functions. Which function is denoted by a particular instance of the name is decided by context. For instance, the operator + can be overloaded to denote integer sum, real sum, and string concatenation. *Coercion*, on the other hand, is a semantic operation that converts an argument to the type expected by a function in a situation that would otherwise result in a type error,

thus allowing the user to omit semantically necessary type conversions (e.g., an integer value can be used when a real is expected).[1] *Value sharing* refers to when a value is shared by several types (`nil` in Pascal is shared by all the pointer types). Finally, in *subtyping* elements of a subrange of an ordered type also belong to superrange types (an element of a subrange of integers also belongs to the whole range of integers).

These extensions however are only exceptions to the monomorphic nature of the type system. Extending a type system in this direction but in a formal and uniform way results in a polymorphic type system. Polymorphic types are types whose operations are applicable to operands of more than one type. [Cardelli & Wegner 85] distinguish two kinds of polymorphism—*universal* and *adhoc*. Universally polymorphic functions work on an infinite number of types, where all the types have a common structure. In terms of implementation, a universally polymorphic function executes the same code for arguments of any admissible type. *Adhoc* polymorphic functions, on the other hand, only work on a finite set of different types, and they may behave in unrelated ways for each type. In this case types may not exhibit a common structure, and they may even be unrelated to each other. In terms of implementation, they may execute different code for each type of argument. Universal polymorphism is considered true polymorphism, whereas *adhoc* polymorphism is considered apparent polymorphism.

Adhoc polymorphism can be further classified into overloading and coercion [Cardelli & Wegner 85]. These correspond to the overloading and coercion mentioned as extensions to monomorphism. Overloading and coercion are not considered true polymorphism because in true polymorphism values can have many types that have some common structure. In the case of overloading a symbol can have many types but the values denoted by the symbol have different and possibly unrelated types. In the case of coercion values need to be converted into

---

[1] [Cardelli & Wegner 85] point out that in untyped and interpreted languages the distinction between overloading and coercion blurs in many situations.

one type before the operator can work on them. Overloading is usually solved by the compiler by giving different names to different functions. Coercion can also be detected during compilation but it can have run-time effects. Some of Napier88's operators exhibit *adhoc* polymorphism. For example, equality and assignment are defined over all types, but the code executed is type dependent. Operators like plus, and minus are also *adhoc* because they are defined over integers, reals, and strings. Although automatic type coercion is not performed in Napier88, the language provides procedures like `float`, and `truncate` to convert integers to reals, and *vice versa.*

Universal polymorphism can also be further classified into parametric polymorphism, and inclusion polymorphism [Cardelli & Wegner 85]. A parametric polymorphic function (also called generic function) has an implicit or explicit type parameter which determines the type of the argument for each application of that function. The language that is used as a model for parametric polymorphism is ML [Milner 83]. ML was built entirely around this style of typing which allows the definition of, for example, a polymorphic identity function which works for every type of argument. Other languages that offer parametric polymorphism are Russel [Demers & Donahue 79], Poly [Matthews 88], and Napier88 [Morrison *et al* 89]. Value sharing is a special case of parametric polymorphism. A kind of parametric polymorphism is offered by Ada with the generic procedure. However, generic procedures are specialized to particular kinds of parameters.

Inclusion polymorphism allows viewing an object as belonging to many different classes that need not be disjoint, that is, there may be inclusion of classes. Every object of a subclass can be used in the context of a superclass. Subtyping is a case of inclusion polymorphism. Objects of a subtype can be uniformly manipulated as if belonging to their supertypes. Inclusion polymorphism can be found in many modern languages (both typed and untyped) like Simula [Dahl & Nygaard 66], Smalltalk (untyped) [Goldberg & Robson 83],

Lisp Flavors (untyped) [Weinreb & Moon 81], and Amber (typed) [Cardelli 84] but not in Napier88. Inclusion polymorphism is also used to model subtyping and inheritance in object-oriented programming [Cardelli & Wegner 85].

Universal polymorphism is a more principled kind of polymorphism than adhoc polymorphism, thus, it was chosen as the kind of polymorphism to be investigated. Both parametric and inclusion polymorphism offer information hiding, and thus, they are both worth investigating to support the analysis.

4.5 Napier88

The programming language used for the experiments was Napier88, [Morrison *et al* 89,Dearle *et al* 90], a general purpose persistent programming language developed at the University of St. Andrews, Scotland. The language was envisaged to be used to construct systems that need to support large amounts of dynamically changing and structured data as is the case in the analysis of AI programs.

Napier88 incorporates most of the abstraction constructs I was interested in investigating, and it does it in a coherent way (which we have seen in section 4.3.3 is important for complexity management): it is a persistent programming language (section 4.4.1) it incorporates eager strong typing (section 4.4.2) **ADT** with a higher order type theory (section 4.4.3) and parametric polymorphism (section 4.4.4). Although research on incorporating inclusion polymorphism in Napier88 was ongoing at the time, it was not available in the version of the language used for the experiments, and thus, it was not investigated. Even with this drawback, Napier88 was chosen for the experiments because it was the only easily available language (environment) that implements, in a coherent way, the other abstraction constructs. It is also part of the Algol family of languages. Thus, the principles of the language are well understood, that is, it is not a strange new language. The new abstraction constructs in the language are also based on good principles (e.g., orthogonal persistence as described in section 4.4.1). There was

also a good communication with the implementors at St. Andrews, which is also very important when using an environment that is still under development.

4.5.1 Consequences for Building AI Programs

Here, I will look at the influence that the choice of abstraction constructs and their implementation in Napier88 has in the building process of a program, and thus in its analysis. We have seen in reviewing the different possibilities (section 4.4) that specific abstraction constructs are usually embedded in a programming language (or environment). This means that there are two possibilities to investigate their use in analysing programs:

- The programming language embodying the abstraction constructs is used both for building the program as well as for its analysis.
- The program is built in some other programming language, and the program is then analysed using the language that embodies the abstraction constructs.

The second choice has the disadvantage that the program has to be translated from one language to the other during the analysis. That is, the analysis would involve, not just understanding what is in the program but, understanding how it was done in the previous language, how it can be done in the new language, and translating it. Another possibility is to translate the program before the analysis. This option is rather unrealistic (unless it is possible to do automatically). Translating a complex program, as those built in AI research, is a major task, and it is unlikely that it would be undertaken just as a pre-step to the analysis. Also if the target language is a strongly typed language, and the original language isn't, some type inferencing will be needed.

Using the same language for building and analysing a program removes the previous problems. It is important, however, to understand how the abstraction constructs can influence the building process. Most facilities or constructs in a language are optional, that is, they may or may not be used when programming

in the language. This is the case of persistence in persistent languages, ADTs, universal polymorphism, and type facilities (for explicit declaration) in weakly typed languages or languages that offer a type inference mechanism. Thus, in general, it is possible to use a language (implementing the abstraction constructs) without using the abstraction constructs during the building process, and only use them for the analysis. However, this is not completely the case in Napier88. Persistence, ADTs, and parametric polymorphism are optional in the sense just described but the typing has to be explicit.

In an explicit strong type system, most types have to be explicitly declared in a program before their instances can be used. Thus, identifying which types are present is likely to be easier during the analysis. It could be argued that it is unusual to have this information in AI programs as they are usually untyped. However, the purpose of my investigations is to find a way of performing the analysis in a structured way, and to see what kind of support is provided by the abstraction constructs investigated. Having type information is likely to make the analysis easier, but it does not solve the problem of how types are used and how they are operated on. Their representational role is still to be identified even in a typed program. Thus, having type information is a helpful starting point. This point is further discussed in section 7.3.6.

4.6 Summary

In this chapter, I have discussed some important aspects of the analysis—information hiding, increased precision, and complexity management. I have identified abstraction constructs developed in Software Engineering that achieve them—persistence, types, ADTs, and polymorphism. I have discussed different computational forms (or kinds) in which the abstraction constructs can be implemented, the different languages in which they are implemented, and argued for the ones chosen for the investigations (i.e., persistent programming, eager strong typing, etc.), as well as the language that implements them and which was used in the experiments—Napier88.

Chapter 5

Experiments

This chapter describes the series of three experiments I performed to investigate the use of Software Engineering abstraction constructs in the analysis of AI research programs, and the development of the Prototype Analysis Method to support the analysis which incorporates the abstraction constructs.

Three dimensions are important in the examples chosen and the program building process of the experiments: *quality of specifications*; *intelligent behaviour*; and *incremental prototyping*. The example problems chosen ranged from a fairly well specified example in the first experiment (a stack) to less well specified examples—a maze game and the *Mu Torere* game. The stack program would not normally be considered to display intelligent behaviour (some problem solving behaviour, for example), whereas the maze and *Mu Torere* programs do, and the *Mu Torere* program is built by incremental prototyping (but not the others). The programs were designed and built by me.

The approach followed in the experiments was a gradual and incremental investigation of the particular abstraction constructs involved, and of the development and testing of the method for the static analysis of AI research programs. The method had to support the identification of the data structures and their operations and the approach chosen was transforming a program (using abstraction constructs) to a form that can be more easily understood on inspection. For a brief summary of the key features of Napier88 syntax relevant to the experiments see appendix C.

5.1 The Stack Experiment

In chapter 4 I discussed the importance of information hiding, increased precision, and complexity management for the analysis of a program. In this experiment I explored the role of Software Engineering abstraction constructs for supporting these important issues in the analysis of a program, an analysis aimed at transforming a program to a form that can be more easily understood on inspection. Here, I will present the objectives of the experiment in section 5.1.1, and the problem to be solved: to replicate the behaviour of a stack, in section 5.1.2. The program built to implement a stack is described in section 5.1.3. This program was transformed to new forms as a consequence of the transformations performed during the investigation of the role of abstraction constructs. I describe these transformations in section 5.1.4, and the Symbol Level description in section 5.1.5. Results and observations from the experiment are presented in section 5.1.6, and section 5.1.7. This is followed by the assessment (section 5.1.8) discussion (section 5.1.9) and outcome (section 5.1.10) of the experiment.

5.1.1 Objectives

In this first experiment the objective was to explore the use and role of abstraction constructs in the analysis of a program. In particular, the objective was to explore the use of abstract data types, polymorphism, and persistence for information hiding, and strong typing for increased precision. This exploration was aimed at transforming a program to a form that can be more easily understood on inspection (but it was not aimed at developing a method to get the new form).

5.1.2 The Problem: A Stack

The example problem chosen was to build a program that implemented a stack. A stack is a collection of data of the same type kept in sequence where data are added to, and removed from, the same end of the sequence. It is easy to

imagine an agent behaving like a stack. From that behaviour, a Knowledge Level description of an agent stack can be derived (see figure 8–1).[1]

This was a fairly well specified problem (see section 5.1.3 for the specification used in this case). The program implementing it did not display what we would normally call intelligent behaviour, and it was not built by a process of incremental prototyping. Although implementing a stack was not a typical example of an AI problem, it was a good example with which to explore initially the use of abstraction constructs. New implemented forms (resulting from applying certain abstraction constructs) of a simple and well specified stack were easy to get. It was also a good example with which to inspect and compare the different new forms, and to see how each abstraction construct affected their interpretation as a stack. For example, it was easy to see if, after applying abstract data types (ADTs) the resulting form of the program could be more easily understood as implementing a stack than the original form.

5.1.3 The Base Program for a Stack

A stack is a well known data structure and thus, building a program to behave as one was a fairly well specified problem. Although more complete specifications of a stack can be found in the literature (see [Thomas *et al* 88, page 32] for an example) I decided to build a simple one for this experiment. I specified a stack as a data structure containing an undetermined number of integers. The operations I defined for it were *createstack* (which creates and returns a new empty stack) *top* (which given a stack returns the item at the top of the stack without modifying the stack) *push* (which given a stack and an item, returns the stack with the item added at the top) and *pop* (which given a stack, returns the stack without its top item). This specification was not complete: it did not contain an operation to check whether a stack is empty or not, for example. However, that did not

[1]Knowledge Level descriptions of this and the other two experiments are given in chapter 8 as part of the discussion of their Symbol Level descriptions.

```
begin
 rec type int_stack is variant(cons:node ; empty:null)
  & node is structure(hd:int ; tl:int_stack)        !type of stack

 let create_stack = proc( -> int_stack) ; int_stack(empty:nil)

 let push = proc(a:int ; s:int_stack -> int_stack)  !stack operations
            int_stack(cons:struct(hd=a ; tl=s))

  let pop = proc(s:int_stack -> int_stack) ; s'cons(tl)

  let top = proc(s:int_stack -> int) ; s'cons(hd)

  writeInt(top(push(3,create_stack())))
end
```

Figure 5–1: Base program for the stack problem. It includes the type of the stack implemented as a combination of `variant` and `structure`, the operations `create_stack`, `push`, `pop`, and `top`. The instruction `writeInt` is included at the end of the program as a simple way of testing the operations `top`, `push`, and `create_stack`, when the program is run and the output is inspected. The testing of these three operations forms the test set used during the transformations. Comments are preceded by the symbol '`!`'.

matter for the purpose of the experiment. The program implemented from this specification sufficed to explore the use of abstraction constructs.

This stack was implemented in the base program (see figure 5–1) as a sequential list of integers with the four operations implemented as separate procedures: `createstack`, `top`, `push`, and `pop`.

Interpreting the base program as implementing a stack requires detailed understanding of the way it is built. For example, there is no abstract data type defining the stack, instead, a list of integers and four independent operations on the list are to be interpreted as implementing a stack. There is little in the program (except for the names used) in the way of supporting such an interpretation. Having to implement the list (and its operations) in terms of the Napier88 types `variant` and `structure` (and their operations) makes the interpretation more difficult. Not only is there no entity stack clearly defined and implemented in the program, there isn't even a clear definition and implementation of the *list* supposedly implementing it (see figure 5–1).

Dynamic Analysis and Test Set The functionality of the program, and thus the test set for it, is very simple. The dynamic analysis of the program involved testing the correct execution of the implemented operations when running the program. For this, two test sets were used which, between them, tested all the operations. The first tested `createstack`, `push`, and `pop`. The second, used as the test set during the transformations, tests `createstack`, `push`, and `top` (see bottom of figure 5-1). This test set does not test all the operations, it does not test `pop`, for example. However, it still serves its purpose as a test set for this first experiment.

5.1.4 Transformations

The abstraction constructs I investigated in this experiment were abstract data types (ADT) polymorphism, persistence, and strong typing. I explored the use of ADT, polymorphism, and persistence to support information hiding, that is, to distinguish between definition, implementation, and use of data structures and operations. Applying each abstraction construct to a program involved transforming it to a new form. Strong typing was used to achieve greater degrees of precision in the definition of the data structures and operations. As strong typing is an intrinsic characteristic of Napier88, it is not possible to transform a program from a non-typed form to a typed one (all programs are typed). Thus, strong typing was explored by investigating the role it played in each of the transformed versions of the program.

Applying Abstract Data Types

The base program was first transformed to a form where a stack was defined as an abstract data type (see `int_stack` in figure 5–2). Unlike the base program, in this new form a `Stack` entity is clearly defined. This form also shows a clear distinction

Definition of the ADT*:*

```
type int_stack is abstype[Stack](create_stack:Stack ;
                                 push:proc(int, Stack -> Stack) ;
                                 pop:proc(Stack -> Stack) ;
                                 top:proc(Stack -> int))
```

Implementation of the ADT*:*

```
let Int_stack = int_stack[list](
                list(empty:nil),                !create_stack
                proc(a:int ; s:list -> list)            !push
                 list(cons:struct(hd = a; tl = s)),
                proc(s:list -> list) ; s'cons(tl),      !pop
                proc(s:list -> int) ; s'cons(hd))       !top
```

Implementation of the type `list`*:*

```
rec type list is variant(cons:node ; empty:null)
& node is structure(hd:int ; tl:list)
```

Use of the ADT*:*

```
use Int_stack as p in
 writeInt(p(top)(p(push)(3,p(create_stack))))
```

Figure 5–2: The ADT `int_stack` defines the type of a stack of integers, and `Int_stack` implements it in terms of the data structure `list`. Stack operations, and the `list`, are implemented in terms of the language types `structure` and `variant`.

between the definition, implementation, and use of a stack, thus, showing the advantage of applying **ADT** for information hiding.

However, although in this new form operations on a stack are defined in terms of the type `list`, they are implemented in terms of the language operations on `structure` and `variant`. A second new form of the program (figure 5–3) defines and implements the operations on the `list` (using language operations on `structure` and `variant`) outside the **ADT**, and uses them in the implementation of the stack. In this new form, the definition and use of stack as an **ADT** are the same as in figure 5–2 but its implementation is clearer: operations on stack are now implemented (as well as defined) in terms of operations on the type `list`, and not in terms of operations on `structure` and `variant` as before.

Implementation of the ADT*:*

```
let Int_stack = int_stack[list](
     list(empty:nil),                              !create_stack
     proc(a:int ; s:list -> list) ; cons(a,s),     !push
     proc(s:list -> list) ; tl(s),                 !pop
     proc(s:list -> int) ; hd(s))                  !top
```

Implementation of the operations on `list`*:*

```
let cons = proc(head:int; tail:list -> list)
     list(cons:struct(hd = head; tl = tail))

let hd = proc(l:list -> int) ; l'cons(hd)

let tl = proc(l:list -> list) ; l'cons(tl)
```

Figure 5–3: Stack operations implemented in terms of `list` operations which are implemented using language operations.

Applying Polymorphism

A stack is a collection of data of the same type kept in sequence. Thus, a stack definition should contain an abstract type of data. This was not the case in the previous forms where the stack contained a particular type of data, integers. When polymorphism was applied to the base program a more abstract data type stack was formed (figure 5–4) thus accomplishing better the definition of a stack.

```
begin
  rec type stack[t] is variant(cons:node[t] ; empty:null)
  & node[s] is structure(hd:s ; tl:stack[s])

  let create_stack = proc[t]( -> stack[t]); stack[t](empty:nil)

  let push = proc[t](a:t ; s:stack[t] -> stack[t])
       stack[t](cons:struct(hd = a ; tl = s))

  let pop = proc[t](s:stack[t] -> stack[t]) ; s'cons(tl)

  let top = proc[t](s:stack[t] -> t) ; s'cons(hd)

  writeInt(top[int](push[int](3,create_stack[int]()))) !test set
end
```

Figure 5–4: A polymorphic form of the base program. The particular type of data is injected during use. The type in the square brackets is the universal quantifier.

Definition of the ADT*:*

```
type int_stack is abstype[Stack](
                create_stack:Stack ;
                push:proc(int, Stack -> Stack) ;
                pop:proc(Stack -> Stack) ;
                top:proc(Stack -> int))
```

Implementation of the ADT*:*

```
 let Int_stack = int_stack[list[int]](
      list[int](empty:nil),                          !create_stack
      proc(a:int ; s:list[int] -> list[int])         !push
       list[int](cons:struct(hd = a ; tl = s)),
      proc(s:list[int] -> list[int]) ; s'cons(tl), !pop
      proc(s:list[int] -> int) ; s'cons(hd))         !top
```

Implementation of the type `list`*:*

```
 rec type list[t] is variant(cons:node[t] ; empty:null)
 & node[s] is structure(hd:s ; tl:list[s])
```

Use of the ADT*:*

```
use Int_stack as p in
 writeInt(p(top)(p(push)(3,p(create_stack))))
```

Figure 5–5: A non-polymorphic ADT stack is defined, implemented, and used. It is implemented in terms of a polymorphic `list`. The particular type (integer) is injected during the implementation of the non-polymorphic stack, and not during use as before.

Combining Abstract Data Types and Polymorphism

The combination of polymorphism and **ADT** was also explored. Polymorphism was applied to the implementation of an **ADT** stack (figures 5–5 and 5–6) and to both the definition and implementation of an **ADT** stack (figures 5–7, 5–8). In figures 5–5 and 5–7, operations on a stack are implemented in terms of operations on `structure` and `variant`, whereas in figures 5–6 and 5–8 they are implemented in terms of operations on `list`.

In figure 5–5 the polymorphic definition and implementation of the type `list` doesn't affect the implementation of the stack (it doesn't make it more abstract) because the stack is defined non-polymorphically. Thus, implementing an **ADT** stack defined non-polymorphically using a polymorphic `list` doesn't make the program clearer. In figure 5–6 a more complete abstraction of the `list` and its operations is achieved, but again, not of the stack.

Implementation of the ADT*:*

```
let Int_stack = int_stack[list[int]](
    list[int](empty:nil),
    proc(a:int ; s:list[int] -> list[int]) ; cons[int](a,s),
    proc(s:list[int] -> list[int]) ; tl[int](s),
    proc(s:list[int] -> int) ; hd[int](s))
```

Implementation of the operations on `list`*:*

```
let cons = proc[t](head:t ; tail:list[t] -> list[t])
    list[t](cons:struct(hd = head ; tl = tail))

let hd = proc[t](l:list[t] -> t) ; l'cons(hd)

let tl = proc[t](l:list[t] -> list[t]) ; l'cons(tl)
```

Figure 5–6: The same non-polymorphic ADT, and polymorphic `list` of the previous figure, with a different implementation. Operations on `list` are defined and implemented separately as polymorphic, and they are used to implement the non-polymorphic operations on the stack.

Definition of the ADT*:*

```
type generic_stack[t] is abstype[Stack](
                    create_stack:Stack ;
                    push:proc(t, Stack -> Stack) ;
                    pop:proc(Stack -> Stack) ;
                    top:proc(Stack -> t))
```

Implementation of the ADT*:*

```
let Generic_stack = proc[t]( -> generic_stack[t])
 generic_stack[t][list[t]](
                    list[t](empty:nil),
                    proc(a:t ; s:list[t] -> list[t])
                     list[t](cons:struct(hd = a ; tl = s)),
                    proc(s:list[t] -> list[t]) ; s'cons(tl),
                    proc(s:list[t] -> t) ; s'cons(hd))
```

Definition of the type `list`*:*

```
rec type list[t] is variant(cons:node[t] ; empty:null)
& node[s] is structure(hd:s ; tl:list[s])
```

Use of the ADT*:*

```
use Generic_stack[int]() as p in
    writeInt(p(top)(p(push)(3,p(create_stack))))
```

Figure 5–7: Polymorphic ADT stack, and polymorphic `list`. Stack operations are implemented in terms of `structure` and `variant`.

Implementation of the ADT*:*

```
let Generic_stack = proc[t]( -> generic_stack[t])
generic_stack[t][list[t]](
   list[t](empty:nil),
   proc(a:t ; s:list[t] -> list[t]) ; cons[t](a,s),
   proc(s:list[t] -> list[t]) ; tl[t](s),
   proc(s:list[t] -> t) ; hd[t](s))
```

Implementation of the `list` *operations:*

```
let cons = proc[t](head:t ; tail:list[t] -> list[t])
 list[t](cons:struct(hd = head ; tl = tail))

let hd = proc[t](l:list[t] -> t) ; l'cons(hd)

let tl = proc[t](l:list[t] -> list[t]) ; l'cons(tl)
```

Figure 5–8: The same polymorphic ADT stack, and `list` of the previous figure, with a different implementation. Operations on `list` are used to implement polymorphic operations on the stack.

A stack can be more easily seen in the form of the program in figure 5–7 because the stack is not just implemented but also defined polymorphically. However, the implementation of the stack is still language dependent. In figure 5–8 a new form is shown where a polymorphic **ADT** stack is defined and implemented in terms of `list` operations (independent of the language operations). It is in this new form that we can clearly see the definition of an **ADT** stack, abstract from the type of data used, and where we can clearly distinguish its implementation in terms of the list data structure. We can also see the implementation of the list in terms of data structures provided by the language.

Applying Persistence

To investigate the use of persistence for information hiding, I applied it to the most abstract form of the program where I had a polymorphic **ADT** stack with the implementation divided in two parts (figure 5–8). I divided this program into three. In the first program, I defined and implemented the stack, and made persistent all the operations involved in the implementation (figure 5–9). It was not possible to make the type definitions persistent in the same way in Napier88 because types are not values.

Persistence of the ADT*:*

```
 use User with stack:env in
begin
 let generic_stack_env =environment()
  in generic_stack_env let cons = cons
 in generic_stack_env let hd = hd
 in generic_stack_env let tl = tl
 in generic_stack_env let Generic_stack = Generic_stack

 in stack let generic_stack_env =generic_stack_env
end
```

Figure 5–9: The same polymorphic ADT, and `list` of the previous figure, without the test instruction. The polymorphic operations are made persistent in the environment `generic_stack_env` of the persistent store.

In the second program the persistent implementation of the stack was used to create an instance of a stack, which I also made persistent (figure 5–10). In the third program, the persistent operations were tested on the persistent instance of the stack (figure 5–11).

Definition of the ADT*:*

```
 type generic_stack[t] is abstype[Stack](
                     create_stack:Stack ;
                     push:proc(t,Stack -> Stack) ;
                     pop:proc(Stack -> Stack) ;
                     top:proc(Stack -> t))
```

Creation of an instance of the ADT*:*

```
use User with stack:env in
use stack with generic_stack_env: env in
use generic_stack_env with
 Generic_stack:proc[t]( -> generic_stack[t]) in
begin
 let stack_one := Generic_stack[int]()
 in generic_stack_env let stack_one := stack_one
end
```

Figure 5–10: An instance of a stack, `stack_one`, is created using the persistent creation operation `Generic_stack`, and made persistent.

Use of the ADT*:*

```
use User with stack:env in
 use stack with generic_stack_env:env in
  use generic_stack_env with stack_one: generic_stack[int] in
   use stack_one as p in
    writeInt(p(top)(p(push)(3,p(create_stack))))
```

Figure 5–11: stack_one is used from the persistent store in the test.

5.1.5 Symbol Level Description

Figure 5–12 presents the Symbol Level description of a stack. This Symbol Level was not obtained from applying PAM to the program (the method had not been developed yet). However, for this simple example, the Symbol Level description can easily be derived from inspecting the form of the program where the stack was implemented as a polymorphic abstract data type, and the three levels of abstraction were distinguished in the program (figures 5–7 and 5–8). Figure 5–7 contains the definition of the ADT, its use, and the definition of the type `list`, and figure 5–8 contains the implementation of the ADT and the `list` operations.

Looking at figure 5–7 we can identify three user-defined types in the program. The ADT `generic_stack` which is (initially) interpreted as defining the user-defined type that represents a stack, `t` which is interpreted as the generic type of the elements of a stack, and `int` which is interpreted as representing the type Integer, which in turn, is interpreted as the type of the elements of a particular instance of a stack created in the program. Thus, in figure 5–12 we can see that the Symbol Level description contains three user-defined types: Stack (short for `generic_stack`), `t`, and `int`.

These interpretations need to be confirmed by inspecting the program. If Stack represents a stack then it should be possible to establish that it represents a collection of data of the same type kept in sequence where data are added to, and removed from, the same end of the sequence.

The polymorphic definition of the ADT in the program (`generic_stack`) ensures that Stack accepts data of any type. That data are of any type is represented by the generic type `t` which also indicates that all the elements of a

User-defined types at the domain level:

Stack: type of any stack

t: generic type for the elements of Stack

int: type Integer, type of the elements of an instance of a stack.

Operations on Stack:

- createStack: $\rightarrow$ Stack
- top: Stack $\rightarrow$ t
- pop: Stack $\rightarrow$ Stack
- push: t $\times$ Stack $\rightarrow$ Stack

Operations on int:

- writeInt: int

Figure 5–12: Symbol Level description of a stack derived (without PAM) from the program where the stack was implemented as a polymorphic ADT, at three levels of abstraction.

stack are of the same type. Thus, the polymorphic definition partially confirms the interpretation of Stack and `t`. The definition, however, does not tell us whether the data is kept in sequence. This is confirmed by the implementation of the ADT. `Generic_stack` in figure 5–8 shows that the data structure representing the type Stack is `list`. `List` is defined (figure 5–7) as a sequence of elements of type `t`. Its recursive definition shows that the length of the sequence is undetermined.

Having established that Stack represents any collection of data of the same type kept in sequence, and that the type of its elements is the generic type t, the next thing is to establish that Stack is operated like a stack, that is, that data are added to, and removed from, the same end of Stack.

In the ADT definition in figure 5–7 we can see that there are four operations defined for Stack—`create_stack`, `push`, `pop`, and `top`. Thus, these operations are described as operations on Stack at the Symbol Level (figure 5–12). Their description includes their parameters and result which specify what is being manipulated in each operation. From the implementation of the ADT and the operations on `list` (figure 5–8) it is easy to establish that data are added to (using `push`), and removed from (using `pop`), the same end of Stack. Data are

also extracted (but not removed) from the same end of the list using the operation `top`, and an empty stack is created with the operation `create_stack`. Thus, from its definition and the way it is operated, we have established that Stack represents a stack. We have also established that `t` represents the elements of Stack. It is the type that is used in the definition and implementation of Stack and its operations to represent the elements of Stack.

In the use of the **ADT**, figure 5–7, we can see that the type `int` and its operation `writeInt` are also present in the program. Thus, they are also described at the Symbol Level (figure 5–12). However, their role is difficult to understand on the basis of the intended functionality of the program. The program was intended to implement a stack, and, from the analysis so far, we have established that the program does indeed implement a stack. However, there is code in the program that is not part of that functionality, and it is as part of this code that `int` and `writeInt` are used. This code basically creates an instance of a stack (an instance of Stack) which is initially empty, and uses it to add (`push`) an element, and to print the element (`writeInt`) after it has been extracted (`top`). In this instance, the generic type `t` is instantiated to `int`. Thus the functionality of this program is that it defines a stack, and it also creates and uses a particular stack of integers.

Given this Symbol Level description, it is easy to understand the control structure that is responsible for the program's behaviour (printing out the integer `3`). This control is implemented in the instruction to use the **ADT** (figure 5–7). The functionality of the user-defined types and operations involved in the instruction are clear from the analysis. Thus, understanding the program's control only involves understanding how the operations being called in the instruction are related.

It should be clear that the Symbol Level description is much easier to obtain (and easier to show how it has been obtained) from the new form of the program (figures 5–7 and 5–8) than from the base program (figure 5–1).

5.1.6 Results of the Experiment

In this experiment information hiding was successfully achieved by applying **ADT**s, polymorphism and persistence. Their application resulted in new forms of the program that were easier to understand as defining and implementing a stack because the data structures and operations are more distinct.

A stack entity was defined abstractly from the type of data it contained which was easier to understand as implementing a stack entity than the original program. I have shown how this kind of information hiding was achieved with polymorphism. **ADT** also provided useful information hiding for improving the understanding of the program. By applying **ADT**, implementational details were hidden from the definition and use of a stack entity. In the forms resulting from applying **ADT**, a stack entity (and its interface) was clearly defined, separately from its implementation and its use. However, even with the use of **ADT**, definitional, implementational, and usage issues were all present in the same program. I have shown how, by applying persistence, these issues were dealt with separately. In the resulting form of the program, see figures 5-9, 5-10, 5-11, it is easier to understand how each issue is treated, because they can be observed separately. This makes the understanding of the whole program easier.

The transformations performed on the program were checked by the use of strong typing. For example, when **ADT** was applied (figure 5–2) `push` was defined before `pop` in the type `int_stack`. If later in `Int_stack`, `pop` was erroneously implemented before `push` (thus changing their order from the stack type definition) the type system would point out the error. Having a strong type system to define, and check, the type of data structures and operations used in the programs increased their precision, and thus, of the interpretations placed on each form as defining, implementing, and using a stack.

5.1.7 Observations

Various levels of abstraction were identified during the transformations of the stack program. In some forms of the program (figures 5–1 and 5–4) we can only see one level of abstraction. In these forms a stack is viewed in terms of types and operations available in the language (`structure` and `variant`). In other forms (figures 5–2, 5–5, 5–7) we can distinguish two levels of abstraction: one in which a stack is viewed as an entity (abstracted from its implementation) and another in which a stack is viewed in terms of the data structure chosen to implement it (`list`) as well as the types available in the language. Finally, we can distinguish three levels of abstraction in other forms (figures 5–3, 5–6, 5–8, 5–9). In the highest level, the domain level, a stack is viewed as a stack entity, independently of the data structure and the language chosen to implement it. In the second level, the structure level, a stack is viewed in terms of the data structure chosen to implement it, in this case, a `list`. Finally, in the third level, the language level, the data structure that implements the stack, `list`, is itself implemented in terms of the types available in the language (`structure` and `variant`).

We can also see important differences in the kinds of information hiding achieved with the different abstraction constructs. When **ADT** or persistence were applied, information was hidden by the separation between definition, implementation, and use. However, when polymorphism was applied, the information hidden was that of the type of data. By abstracting over the type of data, polymorphism provides a different kind of information hiding: *generalization* over the type of data. Thus, two orthogonal kinds of information hiding were achieved.

5.1.8 Assessment

The exploration of abstraction constructs was successful in achieving information hiding, providing increased precision, and in getting new forms of the program that were easier to understand as defining and implementing a stack. This experiment, thus, achieved its objectives successfully.

5.1.9 Discussion

A program can be transformed in many ways, not all of which result in a program that is easier to understand on inspection. The levels of abstraction observed in this experiment (domain level, structure level, and language level) are very interesting in this respect. They suggest a way of organising the transformations performed on a program and the application of the abstraction constructs being investigated. The resulting forms of the program are easier to understand because the new form of the stack is abstracted from implementational details (compare figure 5–1 with figures 5–10 and 5–11, for an example) . These three levels of abstraction subsequently formed the basis of the analysis method which involves transforming a program with the support of abstraction constructs.

The two orthogonal kinds of information hiding observed in this experiment are examples of the different abstractions that can be achieved in a program. However, not both of them are useful for the understanding of specific data structures and operations in a program. Separating definition from implementation and use abstracts from implementational details, and we saw in section 4.1 that this kind of abstraction is important to achieve a form of a program that is easier to understand.

On the other hand, we saw in section 3.3.2 that data structures are usually specialized by the program builder to describe the patterns used in the program to represent particular aspects of the domain, and that during the analysis, it is important to identify these specializations. They provide a clearer semantic interpretation of the data structures and their operations, thus, helping to understand their representational role in the program. Thus, the analysis involves a process of specialization rather than generalization. In this respect, generalization is a kind of abstraction that is not helpful for the analysis. The abstraction provided by the use of polymorphism does not help in the analysis of a program, rather it makes it more difficult as it hides information that we are aiming to find during the analysis. For example, if a list is defined polymorphically, the particular types of data that specialize it cannot be identified until the program is run (when the particular data type, e.g., integer or string,

is injected) and thus, these specializations are not identifiable during the static analysis of the program.

5.1.10 Outcome of the Stack Experiment

The outcome of this experiment was an initial identification of the role played by Software Engineering abstraction constructs in the analysis of a program, and the identification of three levels of abstraction that can form the basis of an analysis method.

I was encouraged to investigate further the use of abstraction constructs such as persistence, abstract data types, and strong typing. They have been shown to provide an appropriate kind of information hiding for the analysis of the stack program. This was not the case for polymorphism.

Further investigation of the successful abstraction constructs also involved the initial development of a method to perform the transformations on a program, a method based on the three levels of abstraction identified during this experiment, and that provides an organised way of performing the appropriate transformations on a program, appropriate in that the new forms can be more easily understood on inspection.

5.2 The Maze Experiment

In this experiment I investigated the use of abstraction constructs to support the analysis of an AI-like program. The abstraction constructs investigated were abstract data types (ADT) persistence, and strong typing, but not polymorphism as it was not found helpful in the stack experiment. Their use was investigated to support the transformation of a program at the three levels of abstraction identified in the stack experiment. The problem chosen for this experiment was a maze puzzle. The idea of using a game, a puzzle in this case, to perform controlled experiments was taken from [Buchanan 88]:

> Small problem areas, sometimes called "micro-worlds", serve as the rat mazes or the Drosophila of AI [McCarthy 83]. The two most widely used micro-worlds have been board games (specifically chess) and the world of children's blocks... In principle, micro-worlds offer the opportunity for controlled experiments; regrettably, the opportunity has seldom been exploited.

I decided to exploit this opportunity by using game problems in my second and third experiments.

5.2.1 Objectives

The objectives of this experiment were to investigate further the use and role of ADT and persistence for information hiding, and strong typing for increased precision during the analysis of an AI-like program. A further objective was to investigate a method to perform the analysis which, based on levels of abstraction and making use of the previous abstraction constructs, provides an organised way of performing the analysis, that is, a way of transforming a program to a form that is easier to understand than the original.

5.2.2 The Problem: A Maze Puzzle

The problem in this maze puzzle is to find a path between two given cells of the maze depicted in figure 5–13, and if it exists, print it. Repetition of subpaths within a path is not allowed.

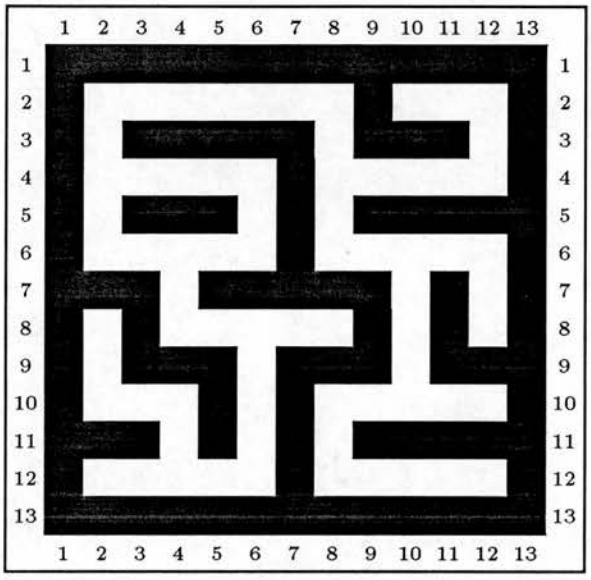


Figure 5–13: The maze. Walls are black and spaces are white. Cells are referred to as (a,b) where a is the vertical dimension and b horizontal.

An agent solving this puzzle would usually be considered as displaying intelligent behaviour (albeit of a simple kind) and from observing it, a Knowledge Level description of the agent can be derived (see figure 8–2). The program to solve the problem was carefully designed and implemented, but a well defined specification was not provided. Although the building process of the program did not involve incremental prototyping, the maze puzzle is a good example of an AI-like problem as it raises problems of representation and inference of the kind typically found in AI problems. Also, the program was more complex than that for the stack, so, more complex transformations could be performed on it. At the same time, its simplicity avoided unwanted development effort, and excessive complexity in the transformations.

5.2.3 The Base Program

The program I built in Napier88 evolved from a program that I had previously built in Prolog to solve this same puzzle. The program in Prolog was an exercise from a knowledge representation course. The specification for the Prolog program (i.e., the exercise) was a drawing of the maze and the description of the problem. The problem was to write two predicates, one to say if a path exists between two points in the maze, and one to print out the path between two points, if one exists. The exercise also indicated the need to think of a way to represent the maze and then defining some predicates with which to form the two required. This description was further expanded for this experiment to indicate that repetition of subpaths was not allowed.

The design of the program in Napier88 involved devising a representation of the maze, and a search algorithm to find paths in it. One way of doing this is to represent the maze as the set of all possible correct paths. I did not consider this design interesting as the problem would be in fact solved by the designer while building the representation of the maze, without the need for further inference in the program. Instead, I chose a more interesting representation for the maze: a sequence of all the cells in the maze where each cell (identified uniquely by a number) has attached to it a sequence of its neighbour cells in the maze (those that share a boundary with the cell). In figure 5–13 we can see that cell (4,8)[2] has three neighbour cells, (3,8), (5,8), and (4,9) whereas cell (8,2) has only one neighbour, cell (9,2). In this way, the maze is represented in terms of its atomic component (cell) and the relation between each component and its neighbours.

The program performs a depth-first search of the maze until a path is found or all the possibilities are exhausted. The process starts by getting the sequence of neighbours of a path's initial cell (which is given by the user). Each of the cells in the sequence is looked at to see if it is the same as a path's final cell

[2]Cells are represented by coordinates here to facilitate their identification in the figure.

(also given by the user). If it is, a path has been found, if it is not, the search process continues by looking at its neighbours. If the search procedure reaches a dead-end, it backtracks until a new cell can be looked at. For example, if we are looking for a path between the cells (2,2) and (2,3) (see figure 5–13) and the program starts the search from (3,2) it will search nearly half the maze, until it backtracks as far as the cell, (3,2). When, at that point, the other neighbour of (2,2), cell (2,3), is looked at, the program recognises it as the final cell of the path, and the path found is printed: [(2,2)(2,3)]. (See appendix D.1 for a listing of the implemented base program.)

Dynamic Analysis and Test Set. The dynamic analysis of this program involved testing that the program found paths in the maze when they existed, and that it wrote an appropriate message when they didn't. For this, several initial and final cells existing in the maze were given, as well as some non-existing ones. The program behaved correctly in all cases, i.e., found the paths when they existed (without repeated subpaths) and indicated their non-existence when they didn't. Out of this, I designed a test set using five cases that I considered interesting: the longest path in the maze, from (12,2) to (12,12) (see left-hand side of figure 5–14 for the path found by the program); a path from (4,4) to (6,4) where these two cells are cells of an island in the maze (see right-hand side of figure 5–14 for the path found by the program); a path with an existing initial cell in the maze: (12,2) and a non-existing final cell: (3,7); a path with an initial non-existing cell: (3,7) and an existing final cell: (12,2); and a path with both cells non existent: (3,5) and (3,7). The program finds a path in the first two cases (without repeated subpaths) and prints a message indicating the non-existence of a path in the last three cases. This test set was found sufficient to test the behaviour of the program, at least for the purpose of this experiment.

5.2.4 Transformations

I performed seven kinds of transformations on the base program, or on transformed forms of it. The transformation map in figure 5–15 reflects the first

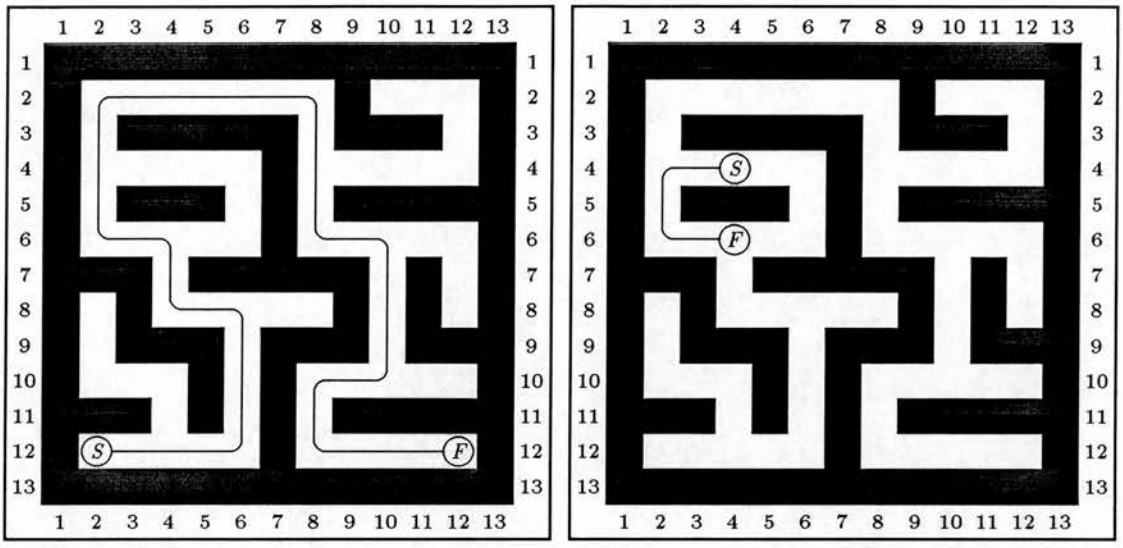


Figure 5–14: Two test paths in the maze.

five, and the transformation map in figure 5–16 the last two. A transformed form was successfully tested when the solutions given by the transformed program to the test set were the same as those of the base program. In most of the transformations (except for 0 and 5) some program forms couldn't be tested as they did not involve finding paths (i.e., the create_maze program in transformations 1, 2, 3, and 6). We can see in the maps that all the forms that could be tested were successfully tested.

A primitive kind of persistence, that provided by data files, was investigated in transformation 0, see transformation 0 in figure 5–15. As a kind of persistence, data files proved not to be satisfactory. The resulting program (initial program in figure 5–15) had to have, and make use of, knowledge of the structure of the data kept in the data file for the program to build the maze correctly. Also, the maze had to be created every time the program was run.

Modularity was applied to the program in transformation 1 (transformation 1 in figure 5–15). As a result, the program after transformation 0 was divided into two modules, one containing all the operations related to the creation of the maze (create_maze program in figure 5–15) and another containing all the operations relating to the use of the maze to find paths (find_paths program in figure 5–15). The program resulting from this transformation was used in

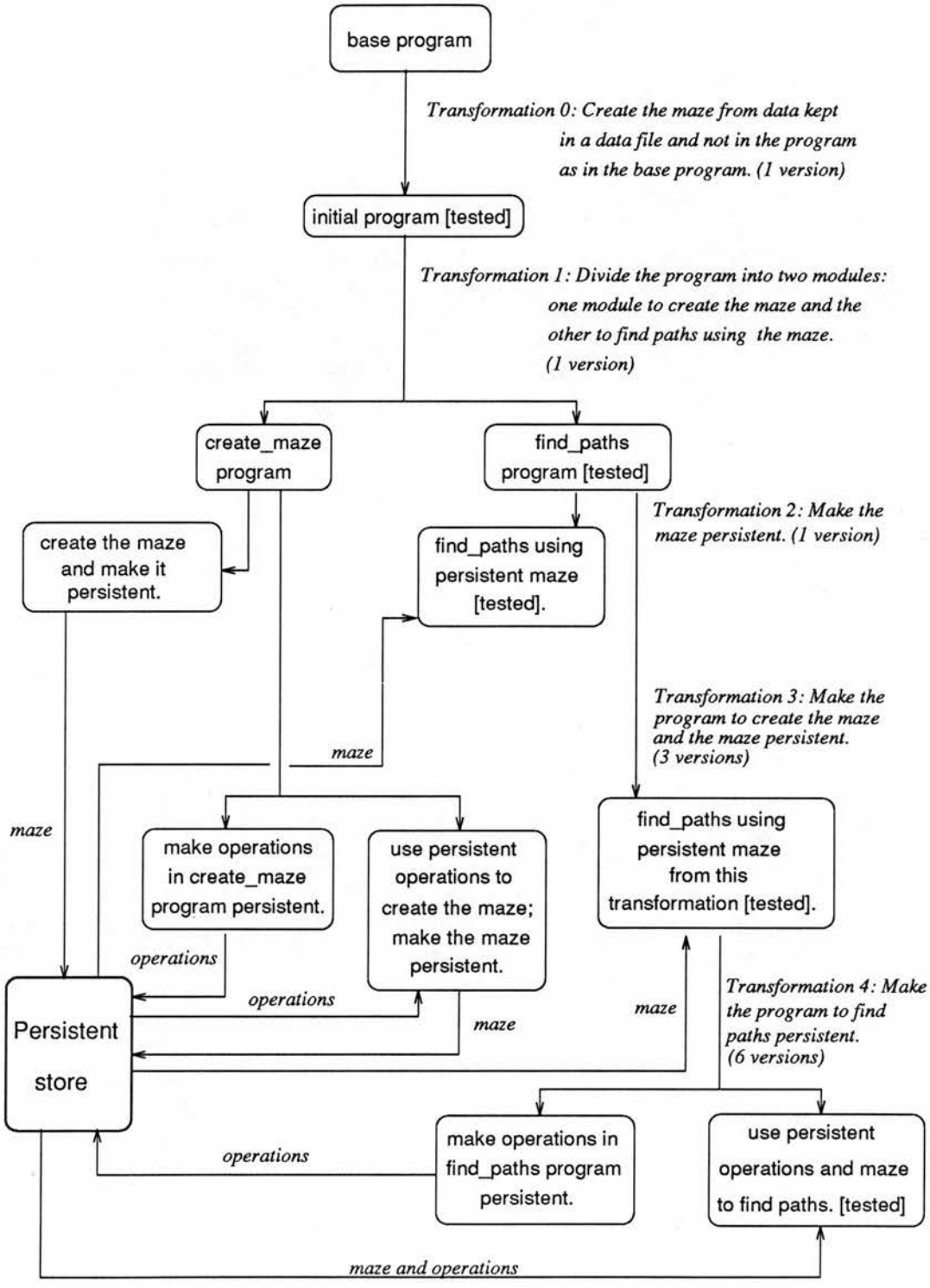


Figure 5–15: First five transformations in the maze experiment.

further transformations to investigate the use of modularity in combination with persistence and transforming a program at various levels of abstraction.

The use of persistence to store data objects, the maze in this case, was investigated in transformation 2 (transformation 2 in figure 5–15) in combination with modularity. The create_maze program (or module) was transformed to contain the operations to create the maze, plus some instructions to actually create it (using the operations) and make it persistent. The find_paths program was transformed to use the persistent maze (after the transformed crate_maze program was executed) to find paths.

In the previous modular forms of the program, the two modules had to be executed together so that the maze created in the first module was used in the second to find paths. This meant that the maze was created each time the full program was run. As a result of using persistence in transformation 2, the two modules could be executed separately. The maze is created only once, and made persistent, when the create_maze program is executed. This persistent maze is used in each execution of the module to find paths.

The use of persistence to store operations (as well as the maze) was investigated in the subsequent transformations. In transformation 3 (transformation 3 in figure 5–15) the operations to create the maze were made persistent in a program, and then used from the persistent store in another program to create the maze. No transformations were performed in the find_paths program (or module). In transformation 4 (transformation 4 in figure 5–15) it is the find_paths module which is transformed into two programs: one in which the operations in the module are made persistent, and another in which they are used from the persistent store to find paths.

Problems and Versions

Transformations 3 and 4 involved more than one version of the transformed program. Most of these versions were aimed at solving problems identified during the transformations. For example, during transformation 3, I had a problem

with the use of persistent global variables. Global variables can normally be manipulated inside procedures without declaring them as parameters or as results. However, when procedures are made persistent, the communication with the persistent procedure is restricted to the interface given by its definition, and global variables are not accessible as before. The first version of transformation 3 did not work because the global variable `maze` was not accessible after it was created. The reason was that the definition of the creation operation did not specify that the result was of the type of the maze. A second version solved this problem by incorporating the type of the maze into the definition of the procedure that created it.

During transformation 4, I encountered more problems involving the use of global variables: the global variable where the path found is kept, and the boolean variable that indicates if a path has been found. In the first version the program executed well the first time, but the rest of the times it just added the initial cell of the new path to the path found the first time. The reason was that the global variables (persistent inside the persistent procedures that manipulated them) were not re-initialized for each execution of the program. In this case the problem could not be solved as in transformation 3; the procedures that manipulated the global variables already had results declared (only one result can be declared for each procedure in Napier88). A second version where the global variables were explicitly re-initialized outside the persistent procedures did not work as the system interpreted them as different variables. In the third version global variables were made persistent, and then manipulated as independent persistent objects inside, and outside, the procedures. This solution worked well, and it was the solution applied in transformations 5 (see section D.4.2) and 6. In the next two versions, which also worked well, two more solutions were attempted involving the initialization of the global variables inside a persistent procedure.

In the stack experiment I said that types could not be made persistent in the current version of Napier88. However, I later found a way to make data types persistent: in the Napier88 system (not in the language) it is possible to access a database, separate from the persistent store, where type declarations can be made

to persist. Two versions were built (from versions of transformations 3 and 4) making persistent data types (in the database) and operations (in the persistent store) and they were compiled using the database. Information hiding was thus more fully achieved when data types and operations were made persistent.

Transformations Five and Six

In transformation 5, the role of persistence was investigated when transforming, at various levels of abstraction, the initial (non-modular) program after transformation 0. In figure 5–16, we can see that user-defined types and their operations from the initial program were defined, implemented, and made persistent in a program, and used from the persistent store in a transformed form of the initial program at three levels of abstraction: language level, structure level, and domain level.

At the language level, as a first step, four user-defined types were identified: two specializations of the language type `structure` (`node1` and `node2`), and two of the type `variant` (`variant1` and `variant2`). Operations on them were also identified from inspecting the code in the program. I call this first step, the identification step. See the declaration of types and persistent objects in section D.2 for their definition and that of their operations (as used from the persistent store). These operations were then defined in a separate file (which I call the definition file). Operations were made persistent by running the definition file. I call this second step the persistence step. The program was transformed into a form that uses the identified user-defined types and operations. I call this step of transformation the substitution step (see section D.2 for the resulting program). This new form was then compiled and executed to validate the identified elements and the transformations. This is the validation step.

At the structure level, the same steps were performed, and two user-defined `lists` (implemented as combinations of `structure` and `variant`) and their operations were identified, made persistent, substituted and validated. See the declaration of types and of persistent objects in section D.3 for their definition; the rest of the program is the transformed program at this level. The same

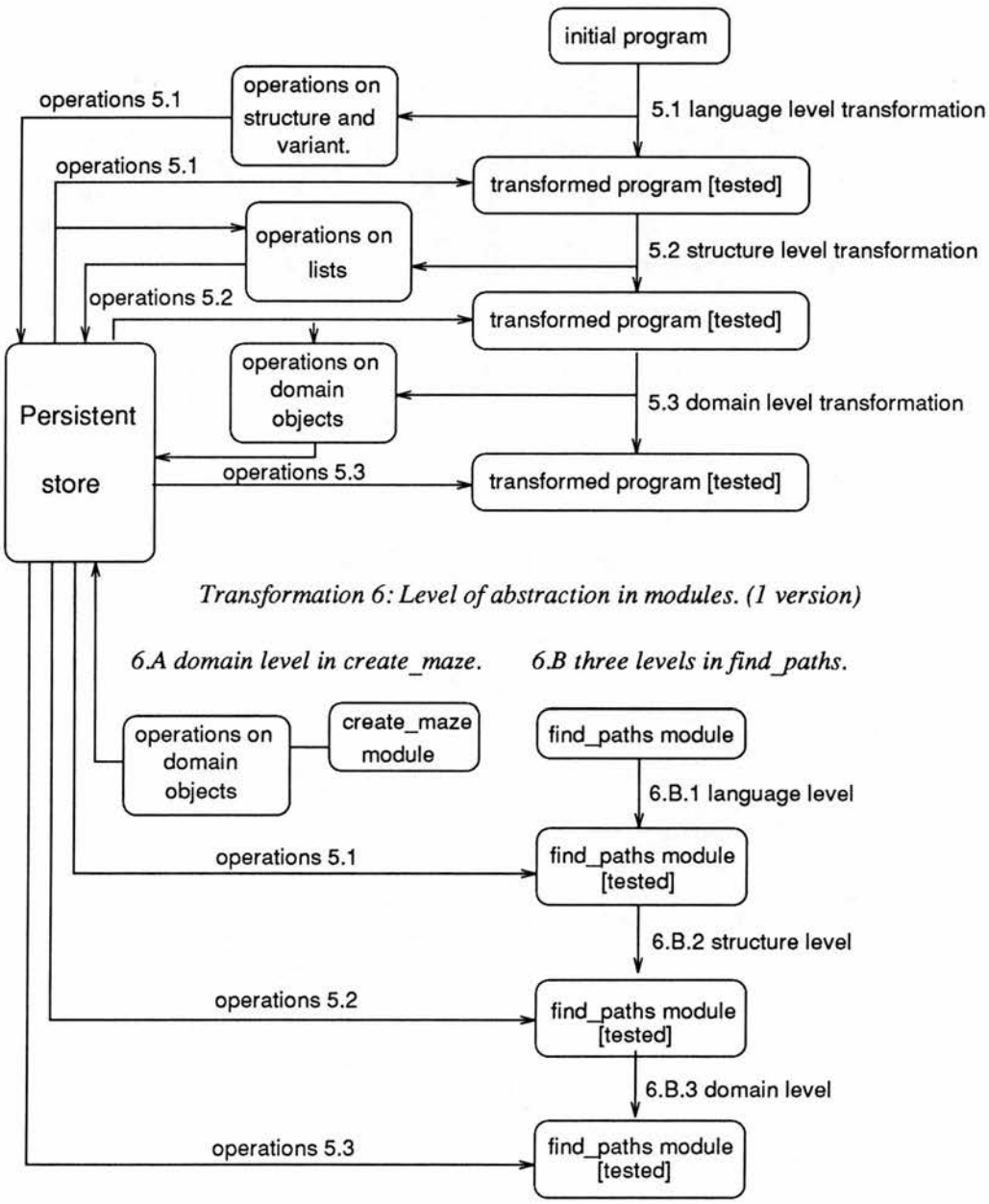


Figure 5–16: Transformations five and six in the maze experiment.

steps were performed at the domain level, and the user-defined types of the domain maze, path, neighbours and cell were identified, as well as operations on them. the resulting program can be seen in section D.4. Figures 5–17 and 5–18 describe the user-defined types and operations identified at the domain level, and which constitute the Symbol Level description of this experiment (explained in section 5.2.5).

In transformation 6 persistence was investigated when transforming, at various levels of abstraction, the modular program resulting from transformation 2. In the first part of transformation 6 (6.A in figure 5–16) user-defined types and operations identified at the domain level in the create_maze module were made persistent. The reason why operations were identified only at the domain level (or for not transforming the module) was that this module could not be tested directly with the test set. It was more interesting to perform a more complete investigation on the find_paths module because it was testable.

During transformation 6.A it became clear that operations made persistent in transformation 5 could be used. Thus, in transformation 6.B (6.B in figure 5–16) user-defined types and operations were identified at each level of abstraction in the find_paths module but they were not made persistent. Instead, persistent operations from transformation 5 were used to transform the module which was validated at each level of abstraction using the test set.

5.2.5 Symbol Level Description

The Symbol Level description in figures 5–17, 5–18, 5–19, and 5–20 was obtained from the domain level analysis in transformation 5 (the user-defined types and operations used for transformation 6 were those identified in transformation 5). I will explain how this domain level analysis was performed and show how the Symbol Level description was obtained.

At the domain level results from the structure level were interpreted in terms of the domain. For this, first domain objects were identified. From the Knowledge Level description (see figure 8–2) three kinds of domain objects were identified:

User-Defined Types:

- maze: representing the maze.
- path: representing paths.
- cell: representing points in maze and path.
- neighbours: representing the set of neighbours of a cell in a maze.

Operations:

Operations for maze:

- Create an empty maze: initialize_maze(→ maze)
- Create a non-empty maze from data store in a data file: create_maze(→ maze)
- Say if a maze is empty: empty_maze(maze → bool)
- Output a maze: print_maze(maze)
- Add a cell and its corresponding set of neighbours to the end of a maze: add_cell_in_maze(cell,neighbours,maze → maze)
- Remove the first cell and its neighbours from a maze: rest_of_maze(maze → maze)

Operations for path:

- Create an empty path: initialize_path(→ path)
- Say if a path is empty: empty_path(path → bool)
- Add a cell at the end of a path: add_cell_in_path(cell,path → path)
- Output a path: print_path(path)
- Reverse the order of the cells in a path: reverse_path(path → path)
- Find a path between two end cells in a maze: find_path(cell,cell,path,maze → path)

Figure 5–17: Symbol Level description of the maze (part one).

maze, path, and cell (or points in the maze). At the structure level `list1` and `list2` had been identified. The initial interpretation at the domain level was that `list1` represented the maze, and `list2` the path. This was based on the declaration of global variables (section D.3). The basic type `integer` represents each cell (when the program is run the messages ask for a number for the initial and final cell of a path).

To check that this interpretation was correct I first inspected the structure of `list1` and `list2` (declaration of types in section D.3). They are both implemented as a recursive combination of the language types `variant` and `structure`. In their definition we can see that `list2` is a list of integers. `List2`

Operations for cell:

- Input a cell: readCell(→ cell)
- Say if a cell is in a path: cell_is_in_path(cell,path → bool)
- Get the first cell in a maze: first_cell_in_maze(maze → cell)
- Get the first cell in a path: first_cell_in_path(path → cell)
- Get the first cell of a set of neighbours: first_cell_in_neighbours(neighbours → cell)

Operations for neighbours:

- Create an empty set of neighbours: initialize_neighbours(→ neighbours)
- Say if a set of neighbours is empty: empty_neighbours(neighbours → bool)
- Add a cell at the end of a set of neighbours: add_cell_in_neighbours(cell,neighbours → neighbours)
- Remove the first element from a set of neighbours: rest_neighbours(neighbours → neighbours)
- Get the set of neighbours of a given cell in a maze: find_neighbours(cell,maze → neighbours)
- Get the set of neighbours of the first cell in a maze: get_neighbours(maze → neighbours)

Figure 5–18: Symbol Level description of the maze (part two).

Figure 5–19: Relations between user-defined types of the Symbol Level.

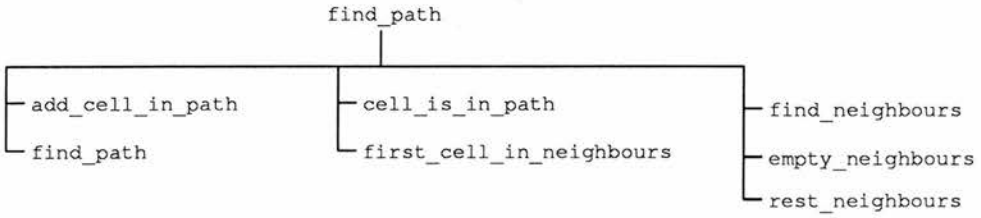


Figure 5–20: Example of relations between operations of the Symbol Level.

then represents a path as an ordered sequence of cells. `List1` is a list of elements, each constituted of an integer and a instance of `list2`. `List1` then represents the maze as a sequence of cells each of which is related to a path. However, this does not make sense, since paths are not supposed to be explicitly represented in the maze. The question is then, does `list2` represent something more than paths?. My interpretation was that it did because (from building the program) I believed the maze to be represented as a sequence of cells where each cell had attached a sequence of its neighbour cells (which I call neighbours). In this interpretation then `list2` represents two user-defined types of the domain: path and neighbours.

This interpretation was further checked by seeing if operations on integer could be interpreted as operations on cell, operations on `list1` as operations on the maze, and operations on `list2` as operations on path and neighbours (or only path in which case the interpretation of the maze would need to be changed). Also it should be possible to use these operations to build further domain operations on maze, path, neighbours, and cell, e.g., an operation to create the maze or to find paths. Section D.4.2 shows how operations at the structure level were interpreted and used to build further ones at the domain level.

The declaration of operations on type cell in section D.4.2 shows how operations on cell were built out of operations on integer, `list1` and `list2`. For example, `readCell` was identified as the operation on integers `readInt`, and `first_cell_in_maze` was defined using `head_list1` which given an instance of `list1` (maze) returns its first `integer` (cell). From this we can see that no operations on `list1` were interpreted as operations on the maze (the same happened with `list2`). The reason is that they involve other types (in the case of `head_list1` it returns an integer) and they can be interpreted as operations on these other types, e.g., `head_list1` was interpreted as `first_cell_in_maze`, an operation on cell, and not on maze. Interpretations were placed by looking at the functionality of the operation. A convention I used which worked well was to interpret operations at the structure level as belonging to the type (at the domain level) of their result (as in the previous case). Sometimes this convention did not help, and the interpretation was based mainly on the functionality of

the operation. For example, I interpreted `member_list2` as the operation on cell `cell_is_in_path` because its functionality is to see if a certain cell is in a list (which can be path or neighbours).

The interpretation that `list2` represents both path and neighbours was shown to be correct by looking at the operations. Some operations on `list2` could not be interpreted as operations on path but they could be (and were) interpreted as operations on neighbours. A good example is the operation `sub_list1` that returns an instance of `list2` (see the declaration of operations on type neighbours in section D.4.2). If it was a path, the `find_path` operation could be defined as a call to this operation. However, we can see in the declaration of operations on type path in section D.4.2 that `find_path` is much more complicated. It uses `sub_list1` (i.e., `get_neighbours` as part of `find_neighbours`) only as the set of cells to explore in each step of the search.

We can now easily see that the Symbol Level description in figures 5–17 and 5–18 is built from the user-defined types and the definitions of the operations identified at the domain level in section D.4. Figure 5–19 shows that the Symbol Level description also describes the relation between the user-defined types. In this figure we can see that the maze is defined as a set of pairs where each pair involves a cell and its set of neighbours, and path and neighbours are defined as a set of cells. Figure 5–20 shows an example of how the operations at the Symbol Level are related. In this example we can see the operations that contribute to the operation (on path) `find_path` (see the body of the operation `find_path` in section D.4.2). The operations are divided in three sets, the two operations on the left are operations on path, the ones in the middle are operations on cell, and the three on the right are operations on neighbour.

5.2.6 Results from the Experiment

In this experiment persistence was successfully applied to achieve information hiding. The transformed forms of the program, all of which were strongly typed, were successfully checked by the type system. This was not the case with abstract data types (ADT). It became clear during the transformations that

its application was not practical in combination with modularity or at different levels of abstraction. In the transformations where the program was divided into two modules (1, 2, 3, 4, 6) **ADT** could not be applied uniformly in each module, as some user-defined types had operations defined across modules. Attempts to apply **ADT** at each level of abstraction in transformation 5 was not successful either, as operations on each user-defined type were identified across levels. For example, some operations on structures were identified at the language level (e.g., indexing operations) but others were identified at the structure level, when those structures were identified as part of lists (e.g., print operations). Thus, it was not clear when all operations on a user-defined type would be completely identified so that an abstract data type for it could be defined, except at the end of the analysis.

During this experiment an initial scheme for an analysis method was successfully developed. This initial scheme (supported by the use of persistence and strong typing) helped to achieve a more abstract form of the maze program. The new form was the result of the identification of user-defined types and operations by incremental transformations at various levels of abstraction. We have seen in section 5.2.5 that user-defined types and their operations were easier to understand in this new form.

I called this initial scheme a transformation process because it involves a process of program transformation. In this scheme the program being analysed is first divided into modules, on the basis of functionalities identified in it, to provide further structure to the program. At each level of abstraction, user-defined types and operations identified in each module are defined and implemented using a strong type system (identification step). Their implementation is hidden using persistence (persistence step) and the program is transformed into a new form that uses the persistent definitions of the identified types and operations (substitution step). The new form is type checked by the type checking system, and its behaviour is tested using the test set (validation step). As different kinds of data structures may be identified in each level (e.g., `structure` and `variant`)

the analysis of a level can be divided into stages, one for each kind of data structure.

5.2.7 Observations

Documentation and Complexity Management. In this experiment documentation and complexity management were identified as important issues in the analysis. It is important, for example, to document the identification step (the user-defined types, and operations identified, and where in the program they were found) as it makes it easier to find them during the substitution step. Also, the analysis at each stage and level is based on the analysis performed at previous stages and levels, and thus on their documentation. This documentation was not performed in this experiment, and although the example program was small enough for this not to cause a problem, it can easily become a problem in more complex programs. The analysis process, that is, identifying, defining, implementing, making persistent, and substituting user-defined types and operations on a program, at different stages and levels, was found to be a complex task whose complexity increases with the complexity of the program. The importance of documentation and complexity management for the analysis will be discussed in section 5.2.9.

Modularity. Dividing a program into modules provides a way of structuring the program based on the functionalities identified in it. It also provides a weak form of information hiding as each module is analysed separately. This helps to understand better the functionality embedded in each one and the relation between its functionality and the user-defined types and operations found in it. For example, the functionality create-the-maze is easier to understand if all the data types and operations involved in it are analysed separately from the rest of the program. It is also easier to understand how each operation contributes to the functionality.

Persistence. As a result of applying persistence objects and processes were more easily identified. A clear example of this in the maze program is the `maze` object. Using persistence, objects (i.e., the maze) and their processes (i.e., creation and use) are taken out of the context of the program. By doing so, their definition and the context of their use is made explicit. As a result, the control flow of the program is made more explicit. This is also the case for global variables transformed into persistent objects that have to be explicitly controlled for their correct use. Persistence also made the analysis process easier in that as operations were taken away from the program into the persistent store, smaller chunks of code to be analysed were left in the program.

In this experiment user-defined types could be made persistent in a separate database provided by the Napier88 system (although not in the persistent store). This lack of uniformity in the application of persistence will be further discussed in section 5.2.9.

Observations On the Method

Identification Step. At the end of the experiment it was still not clear what was involved in performing the identification step. The other steps are more mechanical, and thus, it was easier to understand how they were performed in this experiment. In the identification step, however, it is necessary to understand what is involved in helping a researcher in the identification of user-defined types and operations from a program. This kind of understanding was not completely established in this experiment.

During the analysis, each operation was associated with a single user-defined type (e.g., the operation that creates the maze was an operation on the type of the maze). We have seen that sometimes relating operations to single user-defined types was not so easy (the operation to find paths could be understood as an operation on the type of the maze, use of the maze, or an operation on the type of the path). The functionality of the operations and looking at the result of the operation were the criteria for a decision (e.g., the operation to finds

paths returned a newly created path, and thus, I considered it an operation on the user-defined type of the path).

Substitution Step. The substitution step is a mechanical step which could be performed automatically. However, I soon discovered that automating this step was a complex task. It involved issues such as recognizing different variable names to which an operation was applied, or recognizing different implementations of the same operation in the program. From looking at work on automatic program recognition (reviewed in chapter 3) it was clear that satisfactory solutions to this complex problem were not reasonable to attempt within the scope of this thesis.

Validation Step. The validation step was performed on the basis of the Napier88 type checking system, which checked the transformations performed on the programs, and the use of the persistent user-defined types and operations identified and defined during the analysis. The test set was also used to check that the new form of the program behaved in the same way as the original one.

Levels and Analysis. Analysing a program at various levels reduced the analysis work to be done as the levels progressed. This suggests that levels of abstraction have direct influence on the practical application of the analysis method. If an operation at a certain level was not identified in the program, an operation using it could not be identified at a higher level, and thus, it didn't need to be looked for. For example, if an operation to create a `variant` was not found at the language level, it was clear that a creation operation on a `list` would not be found at the structure level. An operation at a certain level only needed to be looked for in the parts of the program where lower level operations it used were found at the previous level. For example, an operation to create `list1` only needed to be looked for where operations to create its corresponding `variant1` had been identified at the previous level. Finally, if two operations (e.g., `print_list` and `head_list`) used the same lower level operation (e.g., `is_empty_variant`) once one of them was identified (e.g., `print_list`) less work was necessary to identify the other (e.g., `head_list`). This is because only one

instance of the lower level operation (`is_empty_variant`) was to be looked for in the program. The other was already used in the first operation (`print_list`).

A Problem During the Transformations. The original program was mostly implemented in an imperative style, where modifications of an object (including components of list objects) were performed using the assign instruction. For example, the instruction `l'cons(next) := append(...)` modifies the tail of the list `l` to contain the list value resulting from executing a certain `append` operation. When, as a result of the language level, the previous instruction was transformed into `tail2(usage2(l)) := append(...)`, the new instruction produced a compilation error indicating that the assignment could not take place any more. The solution to this compilation error involved changing the assignment operation to a new form that created a new list object (instead of assigning to the old one) where the first element was the same as that of the old list, and the rest of the new list was the result of the `append` operation: `l := value2(create_node2(head2(usage2(l)),append(...)))`. Thus, it seems that performing the transformation process may affect the programming style used (changing it form imperative to functional in this case). This point was further investigated in the third experiment.

5.2.8 Assessment

This experiment successfully achieved some of its objectives: persistence and strong typing were successfully investigated as part of an initial scheme for the analysis of AI programs. This scheme was based on levels of abstraction, and it helped to transform the maze program into a form that was easier to understand than the original program, as the identification of user-defined types and operations made the functionality of the program more clear. The experiment, however, was not successful in identifying a supporting role of abstract data types for the analysis.

5.2.9 Discussion

Complexity Management. Managing the complexity of the analysis process is an issue that can become a real problem in the analysis of more complex programs, thus, it needed to be addressed in the investigation of an analysis method. One way of reducing this complexity is by allowing identified operations to be added, modified, or removed from the set of identified operations in a local and dynamic way. Persistence can be useful for this as it makes it possible to keep an operational store that can be updated locally and dynamically. The use of abstract data types (ADT) on the other hand, would not help in controlling the complexity of the analysis, rather it would add to it as all the ADTs defined at different levels would also need to be managed. For example, in the relatively small program of the maze, it would be necessary to define, implement, and use 10 ADTs, one for each user-defined type. Four ADTs would be defined and implemented at the language level for the two variants and structures. These would then be used at the structure level where two more ADTs would be defined and implemented (`list1` and `list2`) which would then be used at the domain level where four more ADTs would be defined and implemented (`maze`, `path`, `neighbours`, and `cell`). The kind of information hiding provided by ADT (separation between definition, implementation, and use) was also provided by persistence. Thus, it was not clear that the contribution of ADT for information hiding was worth the complexity its use added to the analysis process.

Documentation. Documenting the analysis is very important, as it reflects the researcher's incremental understanding of the program at the various levels. It was not clear, however, how detailed the information recorded should be, or how to define an analysis document to record it. The next experiment should investigate this issue.

Persistence. In this experiment the use of persistence was also seen to be useful to hide chunks of program already analysed. This new dimension of information hiding (different from separating definition, implementation, and use) provided

by persistence also helps in the analysis process as it helps to separate parts of the program already analysed from those that still remain to be analysed.

The impossibility of uniformly applying persistence to data types, as well as to operations, is a inconvenience. Making data types persistent in the database provided by the system is not an ideal solution because it fails to maintain the coherence of the use of persistence.

The Researcher. The role played by the researcher during the analysis is very important. Although the method helps to perform the analysis, it is the researcher (me in this case) who decides on the definitions, and interpretations to be made, not the method itself.

Discussion of the Method

Specifications. In programs built without clear specifications, as is the case in AI experimental programs, it is easy to imagine situations where operations are added or removed during the building process. The current scheme has already been shown to be useful for the analysis of this kind of program. It helps to understand not only what is in the program, but also what is not there (and perhaps was expected to be). It helps a researcher to understand better the program actually implemented in contrast to the one intended or thought to be implemented.

After this second experiment, this method was still an initial scheme which needed further development. The analysis of incrementally built prototypes was still to be investigated. The steps in this initial scheme also needed further investigation, particularly the identification step, and the validation at stages (and not just levels) using test sets.

Test Set. The use of a test set to test the behaviour of a transformed program is very important for the validation step. It is therefore important that test sets for experiments are carefully built to be sufficient. The validation of a transformed program is better performed at each stage (not just at each level) because an

error at a given stage is normally caused by a transformation at that stage. If the validation is performed only at the end of a level, errors from several stages are accumulated, and it is more difficult to identify their origin.

Levels and the Transformation Process. In the current scheme, it is not clear what is defined at the structure level, and what at the domain level. In this experiment, `lists` were identified at the structure level, but the operations identified on them were just the standard operations on lists (i.e., head of a list, tail, member, append, etc.). Other operations that were also performed on lists were identified at the domain level because they were considered more problem dependent. For example, the operation to find paths was identified at the domain level as an operation on the type `path` which was implemented as `list2`, so it was also an operation on `list2`. In general, operations that were identified at the domain level should also have been identified as operations on user-defined types at the structure level. This means that the whole program should be analysed at the structure level, before it is analysed at the domain level. As a result the domain level may not involve looking at code any more, just identifying the representational role of the user-defined types and operations identified during the transformation process.

5.2.10 Outcome of the Maze Experiment

The outcome of this experiment was an initial scheme to perform the analysis of programs based on levels of abstraction and aiming at the identification of user-defined types and their operations. In this scheme, the role of persistence to support information hiding and complexity management was better defined, as was the role of strong typing to check the reliability of the transformations performed, and to increase the precision of the interpretations placed by the researcher on the user-defined types and operations. However, abstract data types were not found to be useful in the scheme. They were not applicable in combination with modularity or across levels of abstraction, and their application

added complexity to the analysis process. Some problems related to the application of the scheme were identified:

- Problems with identifying what information needed to be documented, and with the structure and organisation of an analysis document to record it.
- Problems with complexity management in the analysis.
- Problems with the scheme itself. These included:
  - the need for a domain level analysis separate from the transformation process;
  - the need for a better understanding of the identification and validation steps of the scheme just described;
  - and the need to investigate the analysis of incrementally built prototypes, which was not incorporated into this scheme.

5.3 The Mu Torere Experiment

In this third and final experiment I investigated the use of abstraction constructs to help in performing a complete analysis of an AI-like program. The abstraction constructs investigated were persistence and strong typing (polymorphism and abstract data types were not found helpful for the analysis in the previous two experiments). The initial scheme and the problems identified in the maze experiment relating to its application were further investigated in this experiment. As in the maze experiment, I decided to use a board game problem: the *Mu Torere* game (described in section 5.3.2) for which a program was built without clear specifications, and using incremental prototyping.

5.3.1 Objectives

The objectives of this experiment were to investigate further the suitability of persistence (for information hiding and complexity management) and strong typing (for increased precision) and also an analysis method (following from the initial scheme of the maze experiment) that helps to understand better a program. This involved investigating the analysis of incrementally built prototypes, the analysis at the domain level separately from the transformation process, and the identification and validation steps of the transformation process. The issues of documentation and complexity management were also further investigated.

5.3.2 The Problem: The Mu Torere Game

The *Mu Torere* game [Bell 83] is played mainly by the *Ngati Porou* tribe living on the east coast of North Island, New Zealand, but it can be found among the other Maori tribes. It is the only Maori board game; for this reason, and because the word *mu* is thought to be taken from the English *move*, *Mu Torere* is sometimes held to be derived from Draughts [Bell 83, pages 26, 29]. The board consists of an eight-sided star and a *putahi* or central circle, see figure 5–21. Each of the two players has four *perepere* or pieces of distinctive shape or colour, and the aim of

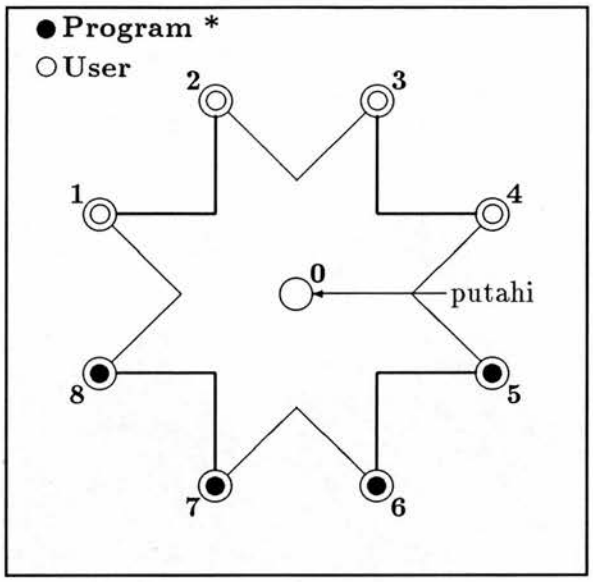


Figure 5–21: The initial configuration of the Mu Torere game (the * indicates turn to play).

each player is the immobilization of the opponent's pieces. The rules of the game are as follows:

1. At the beginning of the game each player places his or her pieces on four adjacent points of the star (see figure 5–21).

2. The players change colours at the end of each game.

3. Black always begins and the players move one piece alternately.

4. There are three possible forms of move:

   a. A piece may move from one of the points to the (empty) *putahi*, provided that one or both of the adjacent points are occupied by an opponent piece.

   b. A piece may move from one of the points to an adjacent empty point.

   c. A piece may move from the *putahi* to an empty point.

5. Only one piece is allowed on each point or the *putahi*.

6. `Jumping` over another piece is not allowed.

7. The player blocking the opponent so that the latter cannot move wins the game.

This game is interesting because it raises problems of representation and inference of the kind typically found in AI problems, and a program playing this game against a human opponent is thus a good example of (intelligent behaviour displaying) AI programs. As the problem is more difficult than the maze puzzle, the complexity of a program to solve it is greater. This was important to investigate complexity management, documentation, and the role of the abstraction constructs and the analysis method when applied to the analysis of more complex programs.

A Knowledge Level description of an agent (say, a human) playing the game is still easily obtainable from observation. It is also the case that an agent playing the game is likely to display an incremental game playing behaviour, i.e., the more an agent plays the game, the better his or her playing is likely to get. In a similar way, two prototypes were built by incremental prototyping for the base program, and they were enough to investigate the analysis of incrementally built programs.

5.3.3 The Base Program

The specification for the program was the description of the problem given in section 5.3.2 together with the scope of each prototype's playing. The scope of the first prototype was to play the game correctly, that is, just following the rules of the game. The scope of the second (which I defined after seeing the behaviour of the first) was to improve the behaviour displayed by the first (i.e., improve strategic aspects of its play) while still playing correctly.

The behaviour of both prototypes was tested with test sets, one for each prototype. Both prototypes contain a log system that kept the games in the test sets so that games could be replayed easily. For each game, the states of the board were recorded, together with the moves, and who had made each move

(the program or the user). Replaying a game from the log involved taking the user's moves from the log, but not those of the program. During the analysis, to check that the transformed programs behaved in the same way as the base program (using the same test set) logged games were compared. Two games were considered equal if the same board states were reached with the same moves played in the same order.

The program was designed as a system of production rules, or rule-based system, [Luger & Stubblefield 89, page 291]. The reason for this choice was that situation-action rules (like those in a production system) can be easily specified for how to play the game, and can also be enhanced to include better strategic playing. The knowledge base in the system is constituted of a set of rules and a fact base describing the state of the game. An inference engine decides on which rule to invoke for each move to be made by the program given a state of the game. The inference process is helped by the use of a working space where partial results of the pattern matching between the rule base and the fact base are kept, and an agenda where possible rules to invoke are kept.

Each rule in the rule base contains a name that identifies the rule. It can have two kinds of patterns: one that can have variables (for the configuration of the board: only one point can be empty but several can be white or black) and another that doesn't have variables (starting condition, program's colour, and turn to move). It also has a test for adjacency of two or three *pereperes*, and a string indicating the name of the rule's action.

Three parts can be distinguished in the inference engine: detection, choice, and deduction.[3] In the detection part, forward-chaining pattern matching is performed between the rule base and the fact base. All the rules whose left-hand sides are satisfied by the fact base are put in the agenda (what is actually in the agenda are the possible moves the program can make for each rule satisfied). In the choice part, the rule to be invoked is chosen from the agenda (the strategy is

---

[3]They are also called pattern matching, conflict resolution, and execution in the literature.

to choose the first one). Finally in the deduction part, the rule chosen is invoked, i.e., its action is executed. Full listings of both prototype programs are presented in appendix E.

Prototype One

In prototype one, the rule base contains eleven rules that represent the game rules, except for game rule 5 (only one piece is allowed on each point or the *putahi*) and 6 (`Jumping` over another piece is not allowed). Rule 5 is implicit in the representation of the board, and 6 is already implicit in the condition for adjacency in rule 4.

Dynamic Analysis and Test Set One. The purpose of the dynamic analysis was to test that prototype one played the game according to the rules of the game, and to see how well it played. For this, I played several games against the program. In these games, I chose a different move in each configuration where more than one move was possible, and I tested that the program followed the rules of the game.

From this dynamic analysis I determined that prototype one accomplishes the purpose it was built for: it plays the game correctly (according to the rules of the game). The program's playing is deterministic, i.e., it always moves in the same way in a given configuration of the board. It also plays very poorly as it makes bad moves in crucial configurations of the board, and as a result, it always loses, even when the user plays badly and the program could win. Given a choice of moves in a given configuration, it always chooses in the following way:

- In configurations where it has the choice to move either to an adjacent point, or from the *putahi*, it always chooses to move from the *putahi* (see figure 5–22 for an example). This is because it processes the rules (and puts the moves in the agenda) in the same order in which they are in the rule base, and the rule for moving from the *putahi* is before the rule to move from an adjacent point.

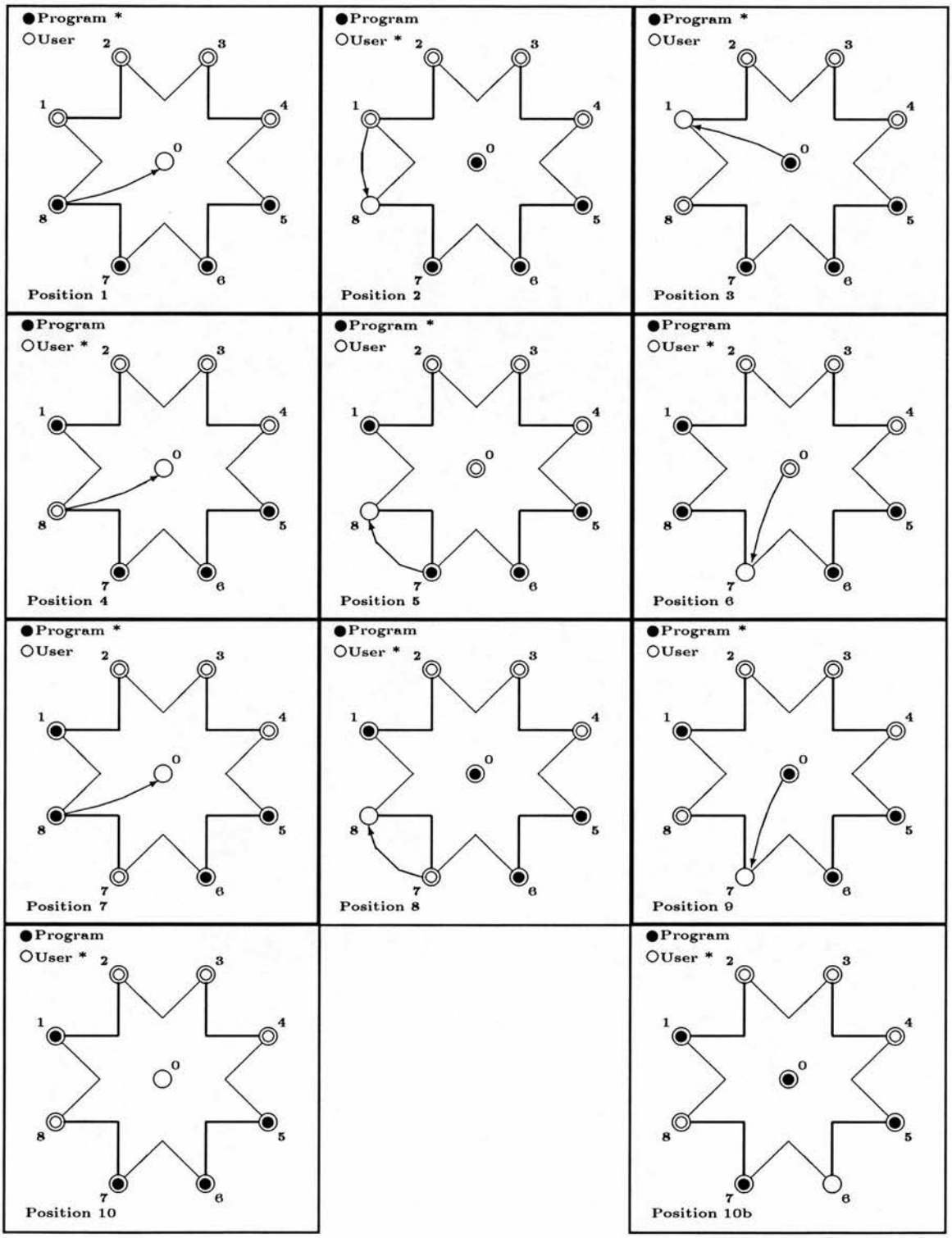


Figure 5–22: A case in which the program doesn't win when it can. If the program had moved from 6 to 7 in position 9 it would have won (position 10b). Instead, the program's move (from 0 to 7) results in the same configuration as in position 4.

- In configurations where it has the choice to move to the *putahi* from different points (up to four) it always chooses to move from the highest numbered point (see figure 5–23 for an example). The reason is that the moves are put in the agenda in a decreasing order of the points they correspond to.

Neither of these behaviours were the result of explicit design decisions. During design it was assumed that all moves in the agenda had the same probability for being good or bad moves, and thus, their order was not considered important. However this proved not to be the case, as the program's behaviour seems to be governed by the way in which the agenda is built. If the order of the moves in the agenda were different, the behaviour would have been different. This means that its behaviour is not just the result of explicit design decisions but also of arbitrary implementation decisions. During testing I also discovered that the agenda contained repeated moves. The reason for this is that a configuration can be satisfied in various ways (two pieces being next to each other) and the same move can be put in the agenda more than once. Checking that a move is not already in the agenda before putting it in avoids this.

As a result of this dynamic analysis I built a test set of ten games (five for each colour) that are representative of the program's playing, and that I used to validate the transformations.

Prototype Two

The second prototype was designed to improve the poor playing of the first. The rule base was enhanced to include some strategic rules, thus providing the program with the means to adopt some strategy for selecting from the possible legal moves in the agenda. Each strategic rule represents a strategy in a certain configuration. There are *winning* and *losing* strategic rules. Winning rules identify on their left-hand side the configurations in which there is a winning next move (one that if performed leads it to win the game) and their actions indicate the next winning move to make. Losing rules identify configurations in

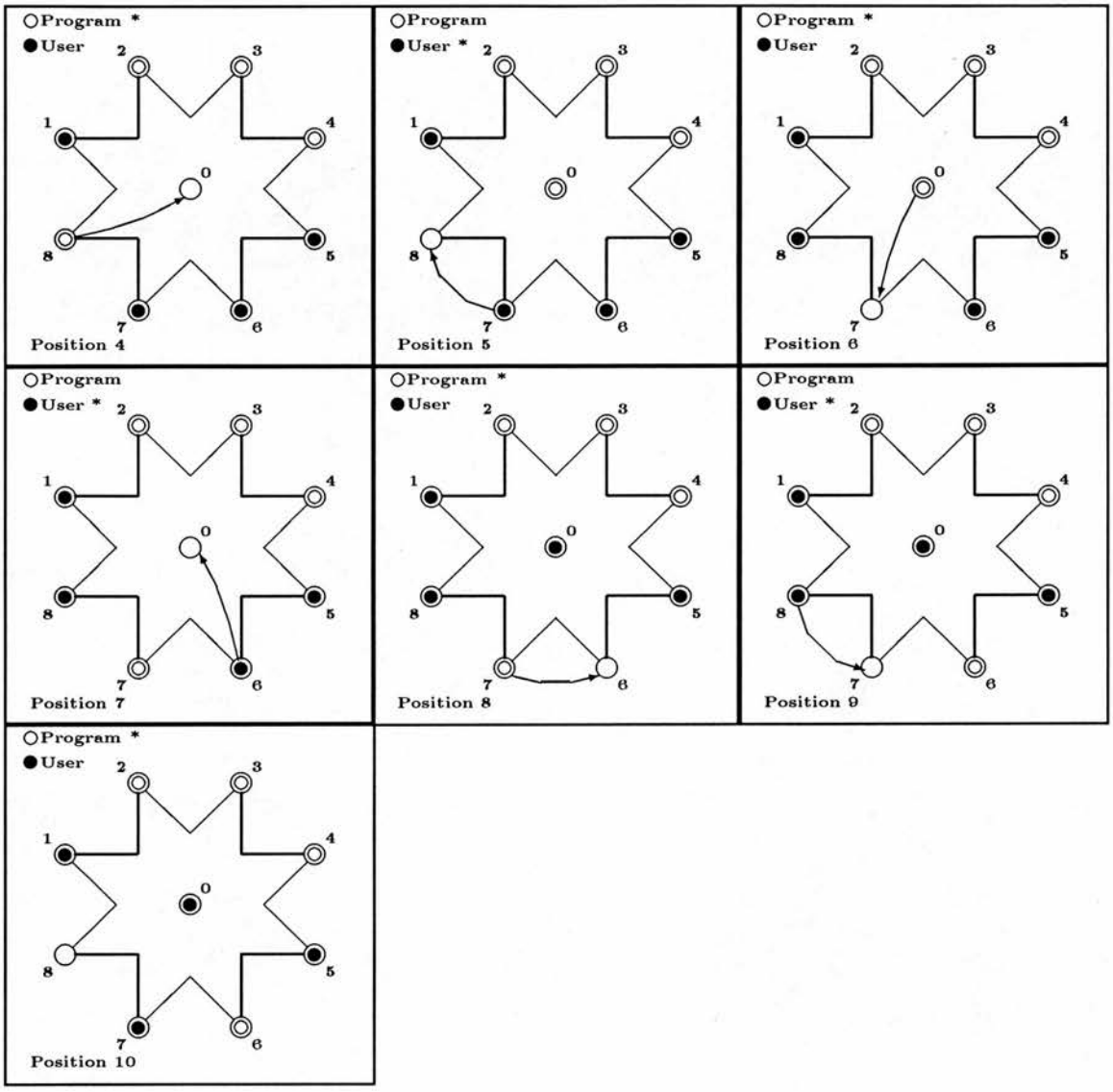


Figure 5–23: An example of prototype one losing as a result of a bad move in the configuration shown in position 4. The user forces the program to play 'no-choice' moves in positions 6 and 8.

which there is a losing next move (i.e., would lead to losing the game) and their actions indicate the next losing move *not* to make.

The strategic rules don't act over the fact base. They seek to restrict the set of legal moves in the agenda to a set of good and legal moves. The new agenda controller applies the constraining strategic moves to the legal moves. When winning moves are applied, all legal moves different from winning moves are removed, and when losing moves are applied, all legal moves that are the same as the losing moves are removed. As a result, the agenda can contain either winning moves, a losing move,[4] or what I call *neutral* moves which are neither winning nor losing moves.

Introducing the strategic rules forces changes in the rule base, and the detection and choice parts of the inference engine. Ten strategic rules were added to the rule base. They implement five strategies repeated for each colour. Each applies to only one legal move. There are winning rules to move to an adjacent point, and to the *putahi* in certain configurations. There are also losing rules to prevent moves to an adjacent point, to the *putahi*, or from the *putahi* in certain configurations. The detection part changes to introduce the new strategic moves (which are different from the legal moves) in the agenda. The choice part also changes to choose a move that is both good and legal.

Dynamic Analysis and Test Set Two. During the dynamic analysis of prototype two I again played several games against the program to test that the program played the game following the rules of the game, and to see how well it played (also in comparison to prototype one). From these games I established that the program plays according to the rules, and that it applies the strategic rules in the appropriate configurations.

The dynamic analysis showed that the program does not lose in the configurations where prototype one did (e.g., in figure 5–23 it moves from point

---

[4]This is the case when a losing move is the only move in the agenda, and therefore can't be removed.

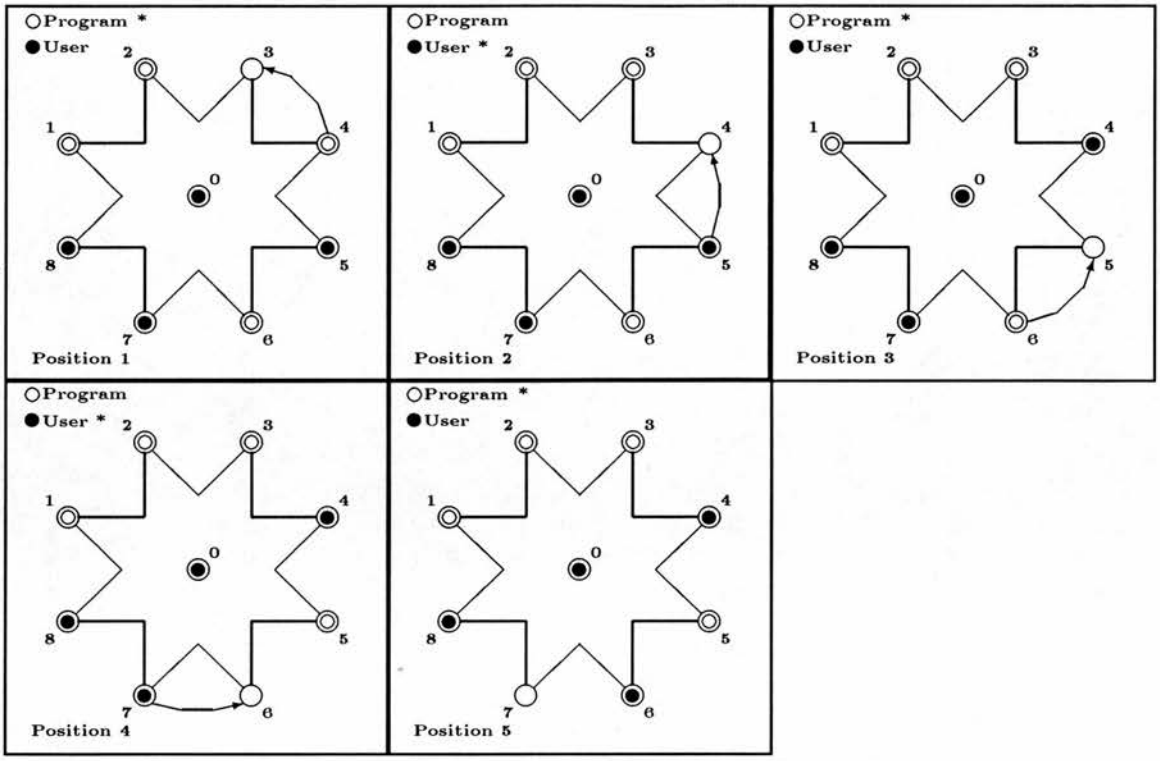


Figure 5–24: An example of prototype two losing after choosing a losing move in position 1.

4, instead of 8, in position 4). It can still lose because there is a losing move that isn't taken into account in the losing rules (figure 5–24).

In the design of prototype two it was assumed that there were no neutral moves that were better than others. However, this was seen not to be true because some can lead to configurations that are more compromising for the user than others (see figure 5–25 for an example). Games three and four of the result test set contain examples of this, and they show that the program does not always choose moves that are better than others in a given configuration. An interesting observation derives from the application of the strategic rule not to move from the *putahi*. This strategic rule is applied in all the configurations where an adjacent move is possible. Its application results in winning moves in certain configurations, but it has the problem that it is also applied in certain configurations where a move from the *putahi* is a better move than an adjacent move.

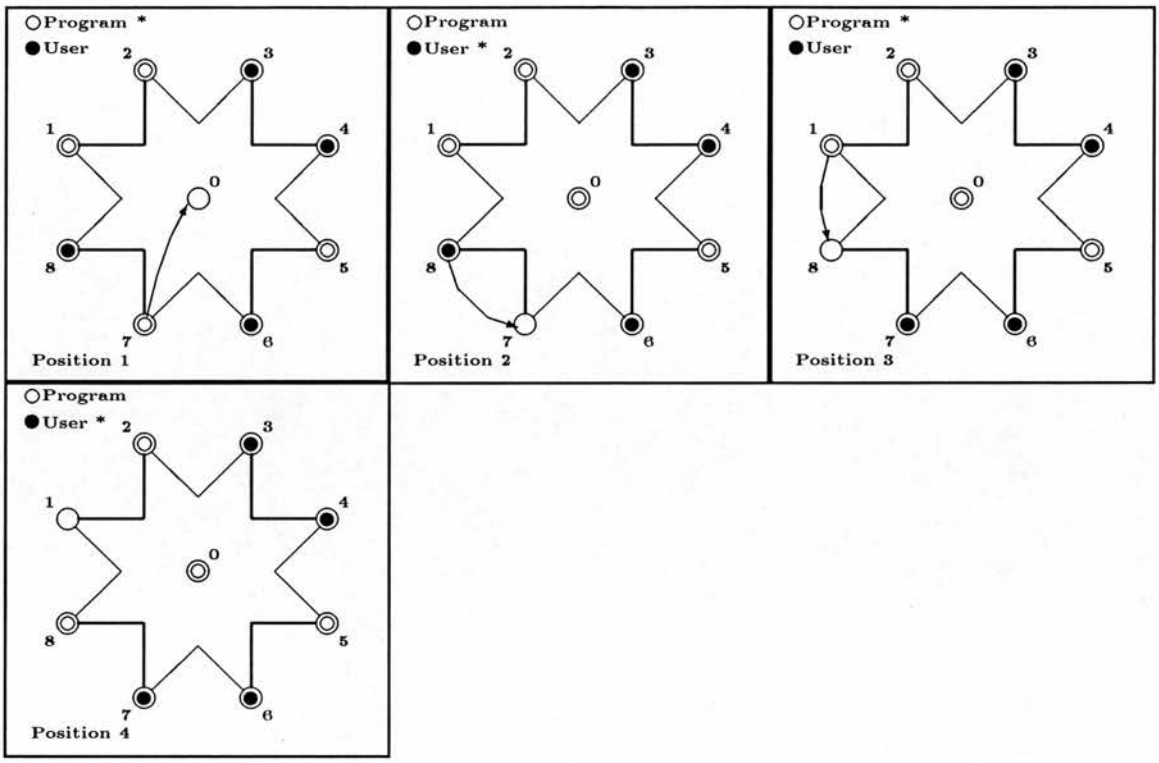


Figure 5–25: An example of prototype two making a better (neutral) move. Moving *pereperes* 5 or 7 in position 1 is better than moving 1 or 2.

The dynamic analysis also showed that the winning rule to-move-to-an-adjacent-point is redundant because the losing rule not-to-move-to-an-adjacent-point satisfies the same configurations, plus some others.

A test set of twelve games was created from this dynamic analysis. The six games in which the program plays with white *pereperes* are the following:

- Game one confirms the application of the losing strategic rules not to move to an adjacent point or from the *putahi* in the appropriate configurations.
- Game two confirms the application of the strategic rule not to move to the *putahi* in the appropriate configuration.
- Games three and four test the scope of the program playing.
- Game five confirms that a user understanding the program's game can easily win.

- Game six confirms the use of the two winning rules in the appropriate configurations.

The set of games in which the program plays with black *pereperes* differs in that game one confirms the application of the losing rule not to move from the *putahi* and game six confirms the application of all other strategic rules.

5.3.4 Transformations

The transformations involved the complete static analysis of the two prototype programs: the division of the program into modules, the application of the transformation process, and the analysis at the domain level (figure 5–26). During the analysis new versions of the programs were obtained (figure 5–27).

5.3.5 Analysis of Prototype One

To structure the program and separate distinct functionalities, the base program was divided into nine modules:

1. A *types module* containing all the data types declared and the initialization of global variables.

2. A *printing module* with all the print operations.

3. A *log module* containing the log system to keep the games played.

4. A *rule base module* containing the set of rules, together with the implementation of the test functions of the left-hand sides of the rules, and the implementation of their actions.

5. A *detection module* implementing the detection of the rules that are satisfied by the fact base at a given point.

6. A *pattern matching module* implementing the pattern matching of each condition of a rule with the fact base.

7. A *choice module* implementing the strategy used to choose the rule to be executed.

8. A *deduction module* for the execution of the rule chosen.

9. A *main program module* for the main program.

Transformation Process

The transformation process involved transformations at two levels of abstraction: language level divided in three stages, and structure level divided in six stages (figure 5–26). In each stage the analysis focused on one particular kind of data structure. For example, stage one of the language level focused on structures, and stage four of the structure level focused on complex lists. At the language level, the user-defined types identified were specializations of structures and variants, as they were the only complex language types found (no vectors were found, for example). Based on them different kinds of lists and other data structures were identified (stages three and five of the structure level).

During the analysis, names of identified user-defined types were changed to incorporate the understanding of the program at each level. For example, a type that in the base program was named `board`, at the language level was renamed as `variant1`, which, at the structure level, was renamed as `list1`. See figure 5–26 for a description of the user-defined types and operations identified at each stage.

Four steps were performed in each stage: identification, persistence, substitution, and validation. The identification and substitution steps were done in each module separately, thus making use of the new structure of the program (see figures 5–28 and 5–30). Records of these steps were kept in the analysis document. All the operations identified in the various modules were made persistent in the persistence step (see figure 5–29) and the transformed program was validated making use of the strong type system and the test set (see figure 5–31). See figure 5–27 for the resulting program and the contents of the persistent store at each stage, and section E.3 for the final form of the program.

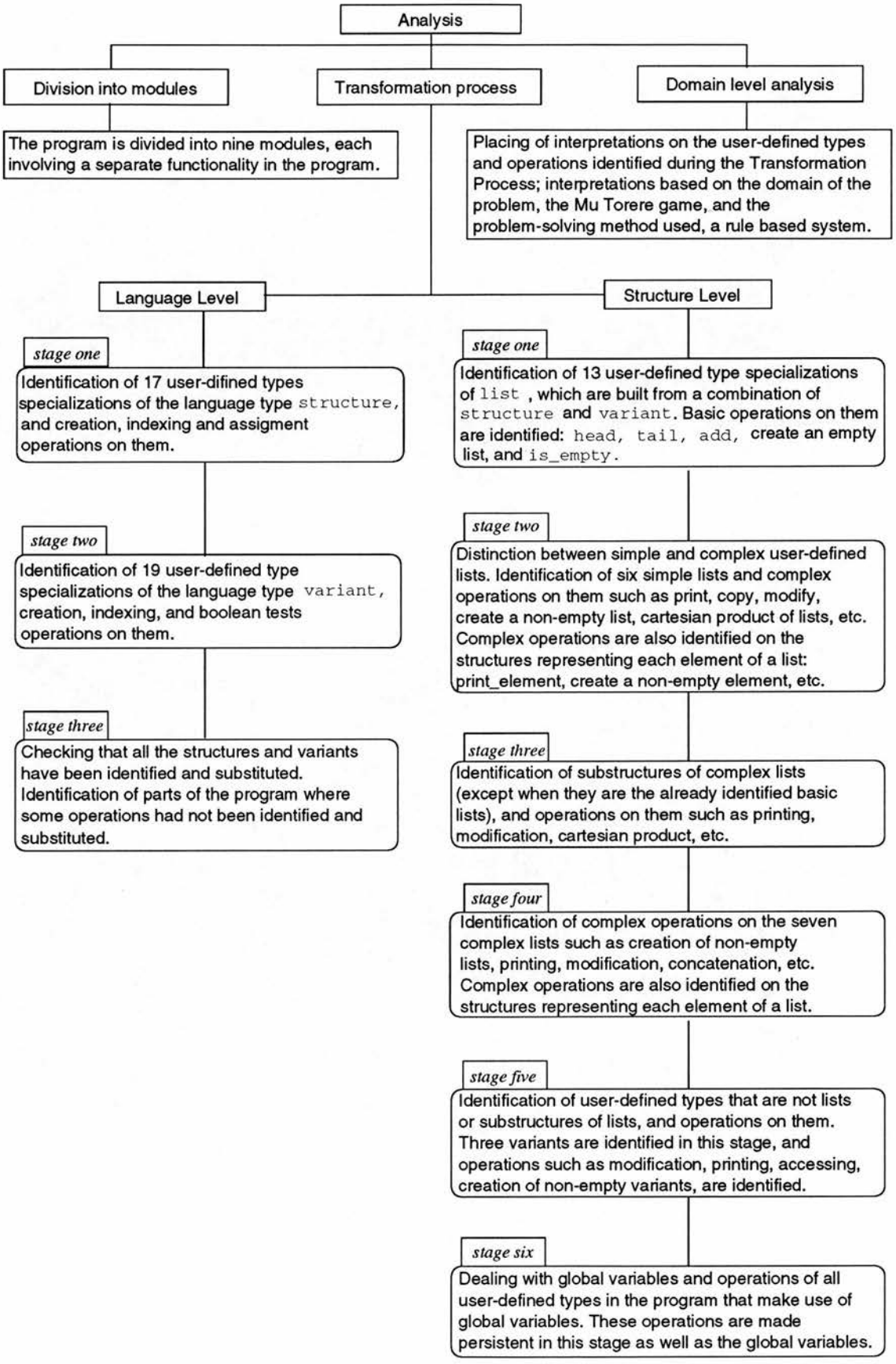


Figure 5–26: Steps of the transformations performed on prototype one and two.

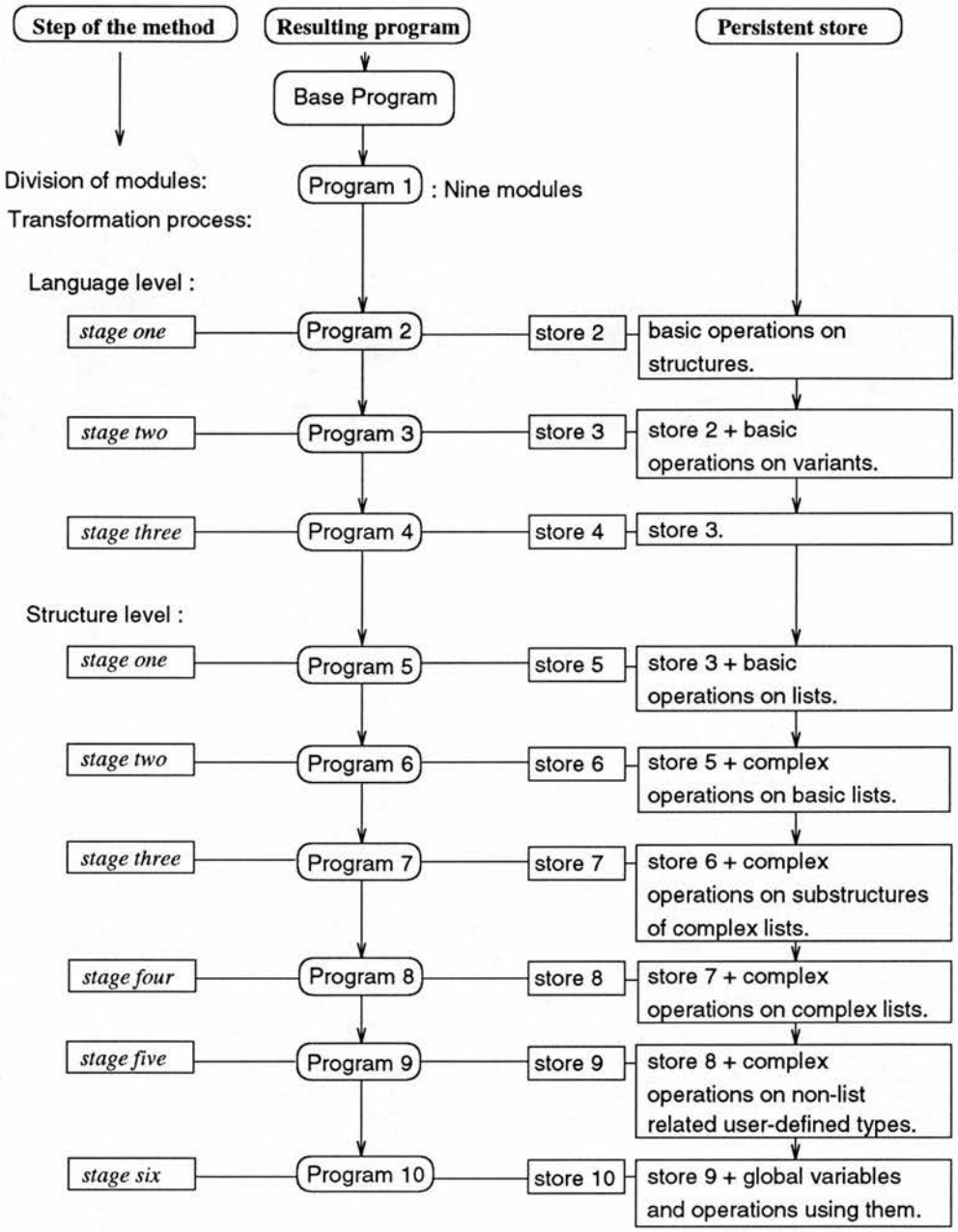


Figure 5–27: Transformed programs and corresponding persistent store during the application of the method.

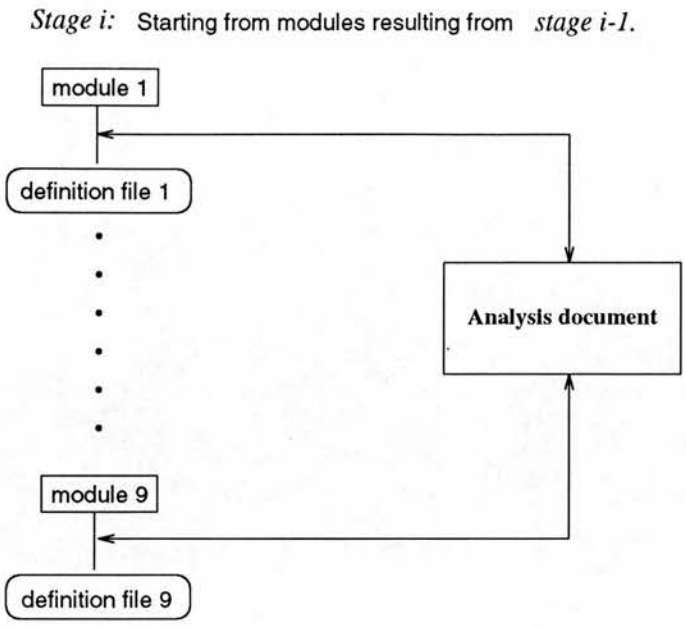


Figure 5–28: Identification step in each stage.

The identification step was mainly based on inspection of the code. For this, old names of types and their fields were looked for. Names of fields helped to identify indexing operations on the fields named. Names of variables were not used for this since one name could be used for different instances of several user-defined types. Modules were useful to focus the inspection on smaller chunks of code, and to use the particular functionality of each module guided the inspection. For example, a type representing the rules was quite likely to be found in the rule base module. All the operations identified in each module were defined in a definition file (see figure 5–28) which formed the basis for the persistence step. The bidirectional arrows in figure 5–28 indicate that information in the analysis document (from previous stages and levels) was used during the identification, and that results from the identification were recorded in the analysis document.

In the persistence step (figure 5–29) operations were made persistent in the persistent store, and user-defined types in the database.[5] The definition file for a stage was the result of the union of the definition files for each module

[5]As the persistence of user-defined types in the database has already been discussed, I won't consider this distinction further in the text.

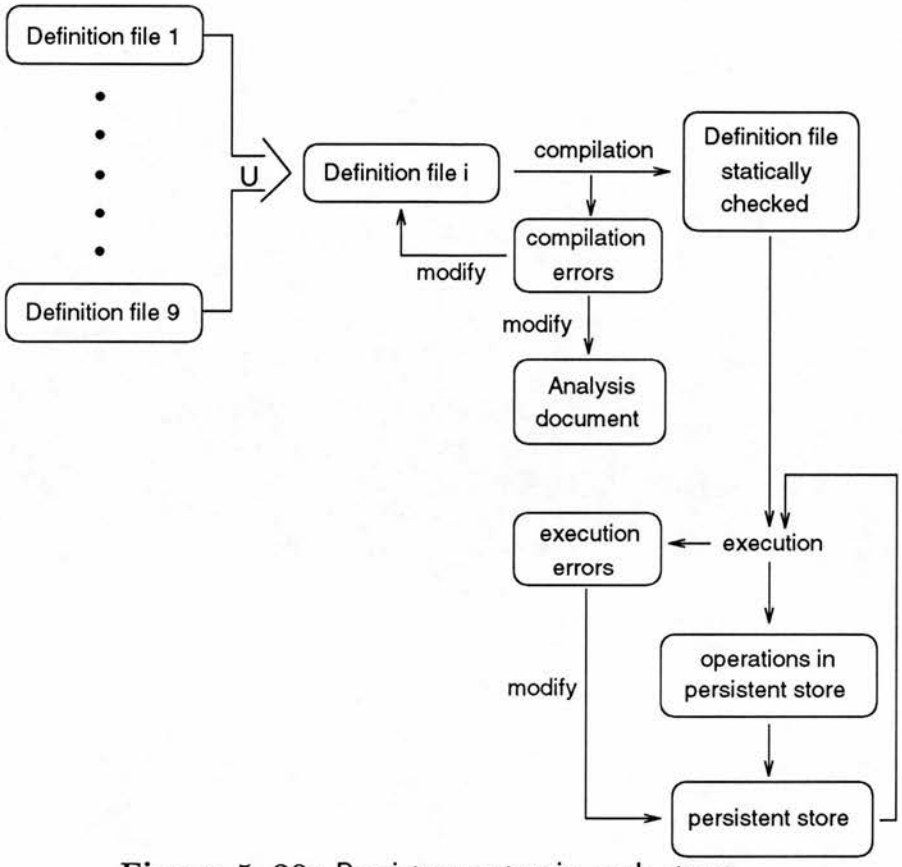


Figure 5–29: Persistence step in each stage.

because operations could be identified in various modules. Solving compilation errors involved re-defining or re-implementing some of the operations in the definition file. These modifications were reflected in the analysis document. When the compiled definition file was executed the operations were made persistent. If execution errors occurred, they only involved the persistent store (not the program): an operation was already persistent, or an operation invoked from the persistent store was not present, for example.

The substitution step was guided by information in the analysis document (figure 5–30). This information had been added during the identification step, and it included the user-defined types and operations identified and where in the modules they were found. This information helped to find what was to be substituted and where.

In the validation step the transformations were checked by the type system, and the behaviour of the transformed program was tested with the test set (figure 5–31). For this, modules were concatenated to form the complete program

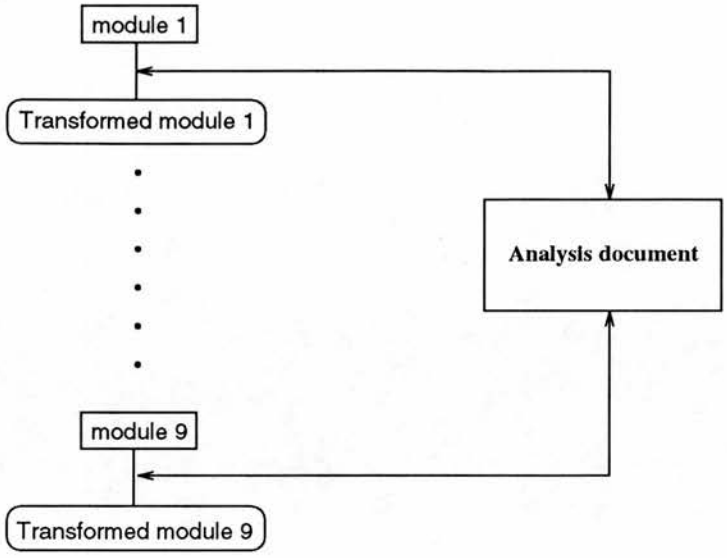


Figure 5–30: Substitution step in each stage.

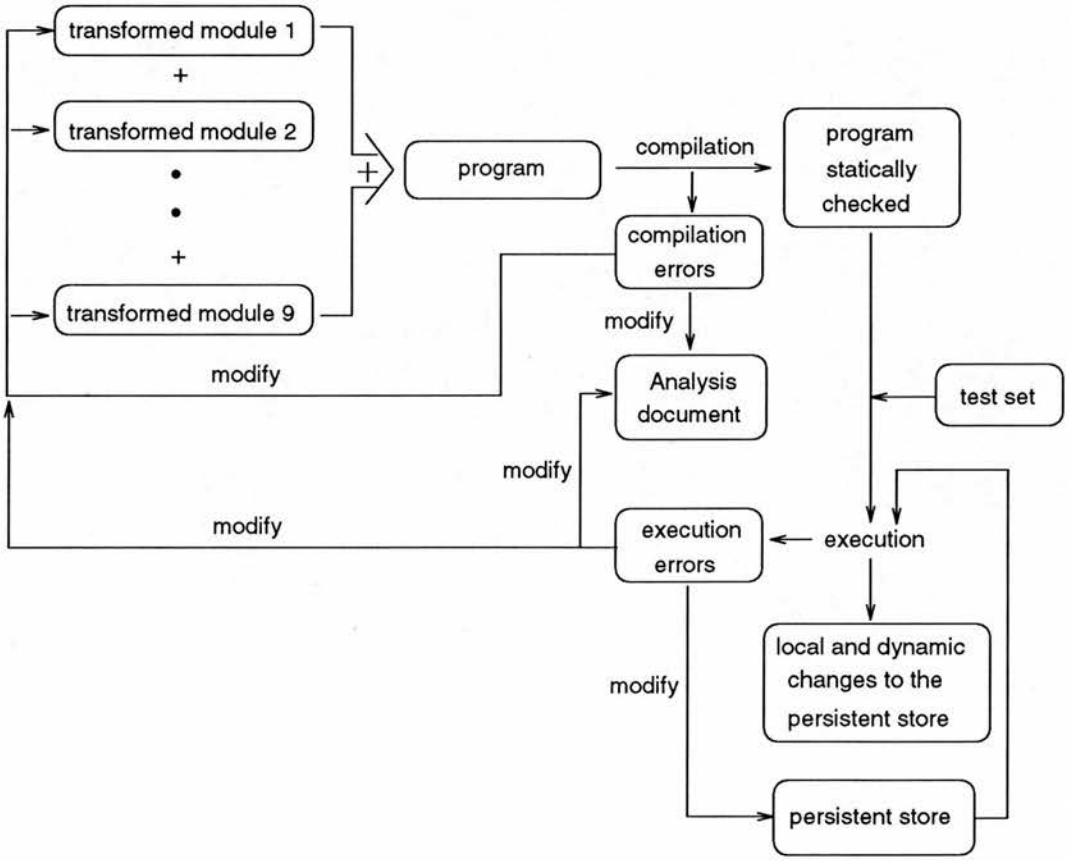


Figure 5–31: Validation step in each stage.

(incremental compilation is not available in the version of Napier88 used). The type system performs the checking both statically (compilation) and dynamically (during execution) which includes checking the correct use of the operations in the persistent store. Sometimes several iterations of the compilation and execution cycle were necessary (see figure 5–31).

To solve compilation errors modules were modified (not the complete program) because modules are transformed at each stage and not the final program. Errors during execution could involve changing modules or the persistent store. For example, an error could be that the definition of an operation was different in the program and the persistent store. When the error was in the definition used in the program, the corresponding module had to be changed and the whole process of validation had to be re-started. When the error was in the persistent store, the solution involved re-defining the operation in a definition file (separate from the final definition file of the persistence step) and modifying only that operation in the persistent store. The local and dynamic nature of the persistent store enables this latter kind of changes to be made in this way.

Illustrations of user-defined types (and their relationships) identified at different levels of the transformation process can be seen in figures 5–32 and 5–33.

Starting from the bottom, structures and variants identified at the language level form the basis of lists such as `list6`, `list8`, `list10`, and `list12` (figure 5–32) or `list3`, `list4`, and `list5` (figure 5–33) which are built out of structures and variants with the same numbers at the structure level. They can also form the basis of substructures of complex lists. For example, `structure7`, `structure9`, `structure11`, and `variant11` are substructures of `list12` (figure 5–32) and `variant7` and `variant2` are substructures of `list5` and `list4` (figure 5–33).

Operations for each user-defined type were identified at various stages, e.g., basic operations on `list12` were identified at the first stage of the structure level, and more complex operations at the fourth stage (see figure 5–26). Figure 5–34 illustrates the operations on the types in figure 5–33: an operation to create an instance of `list5` requires a creation operation for `structure5` which is its

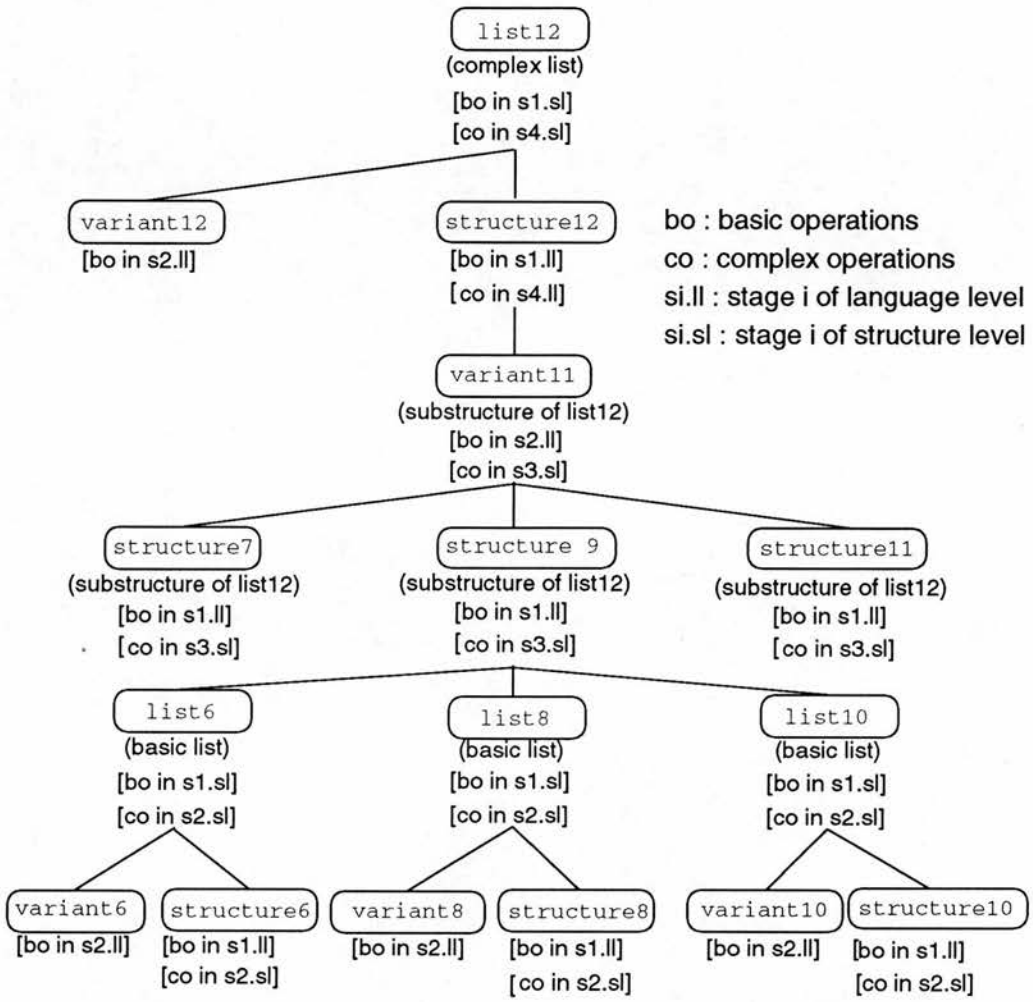


Figure 5–32: Illustration of user-defined types related to `list12` and their relationships.

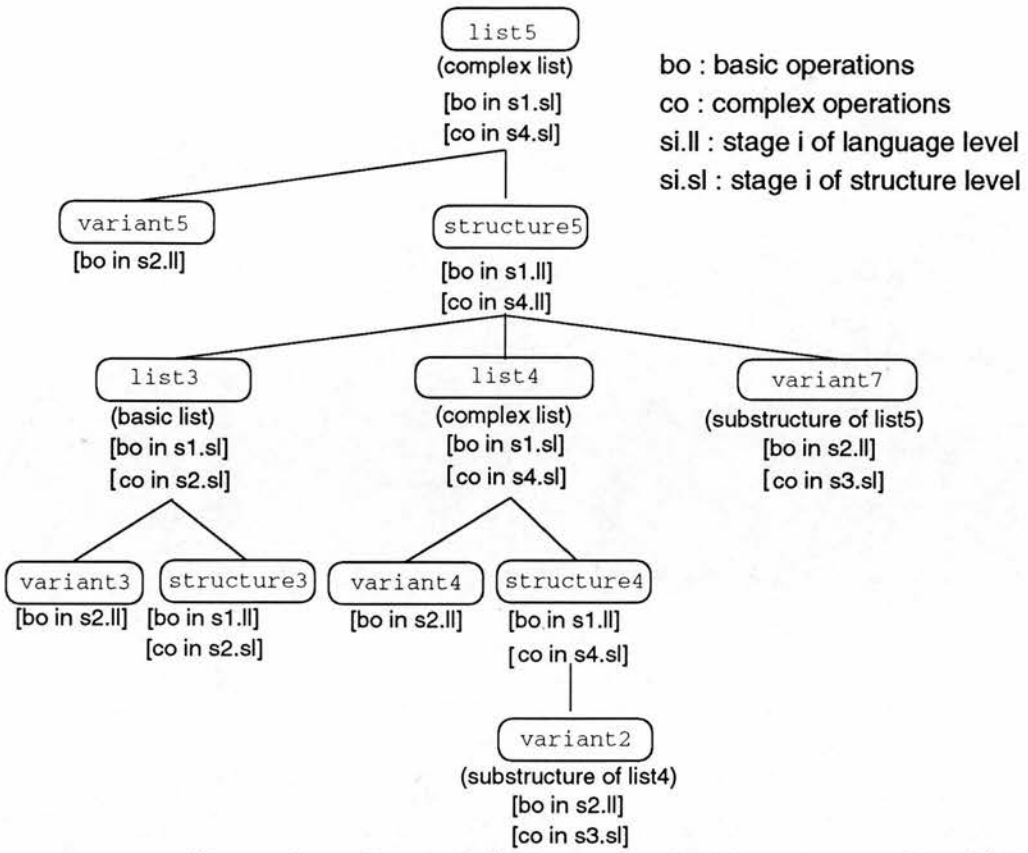


Figure 5–33: Illustration of user-defined types related to `list5` identified at different levels and stages.

component. This in turn uses operations on `list3`, `list4`, and `variant7`. See figure 5–35 and figure 5–36 for the tree of dependencies for the creation operation of `list3` and `list4` respectively. See also section E.3.2 for the description at the structure level of `create_list5` and the one of the creation operations on `structure5`.

Domain level Analysis and Symbol Level Description

At the domain level, interpretations were placed on all the user-defined types and operations identified during the transformation process. They were interpreted as data structures and operations of the problem domain (*Mu Torere* game) or the problem solving method (rule-based system). The Symbol Level description of this experiment was obtained as a result of performing these interpretations and their subsequent checking. To illustrate this process I will explain the domain

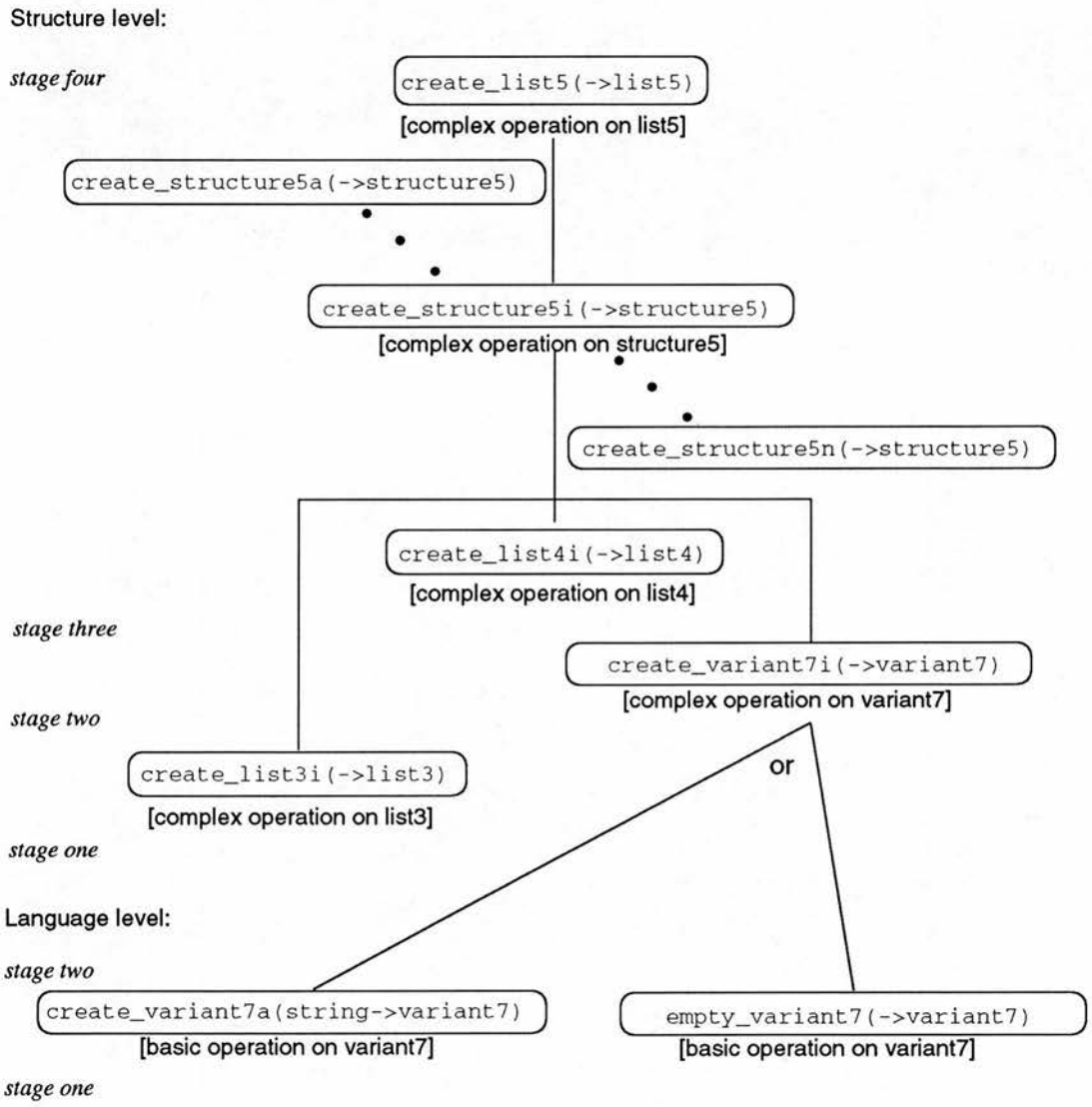


Figure 5–34: Illustration of operations related to `list5` at various stages.

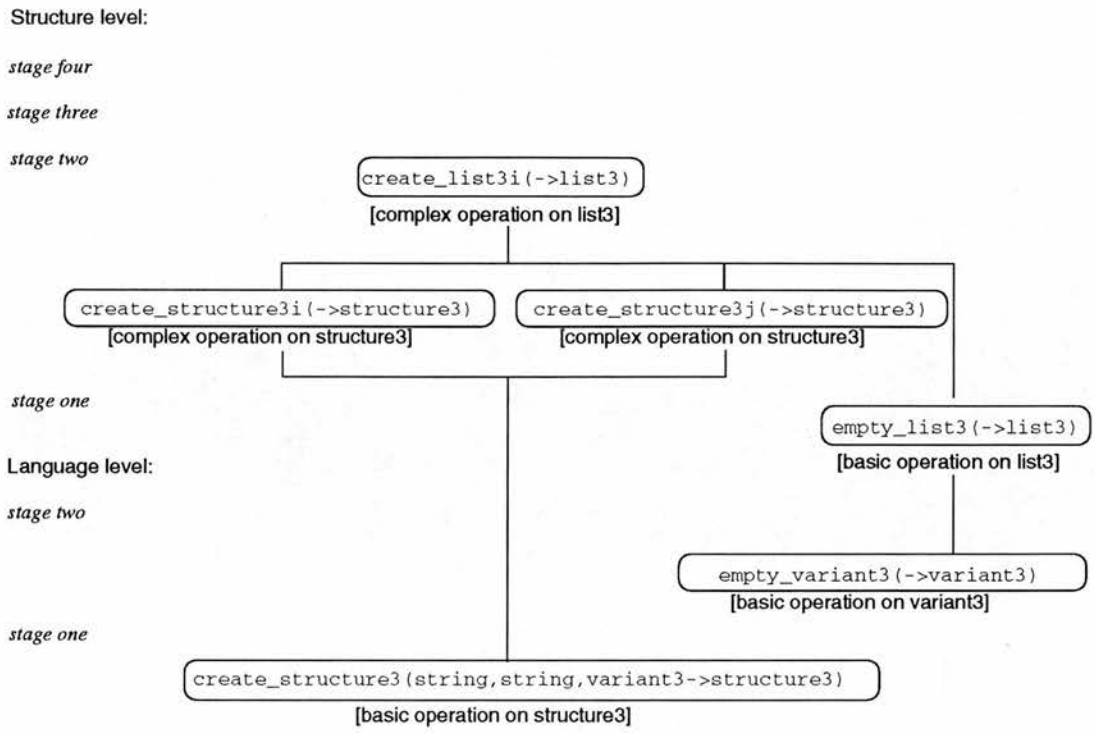


Figure 5–35: Illustration of operations related to `list3`.

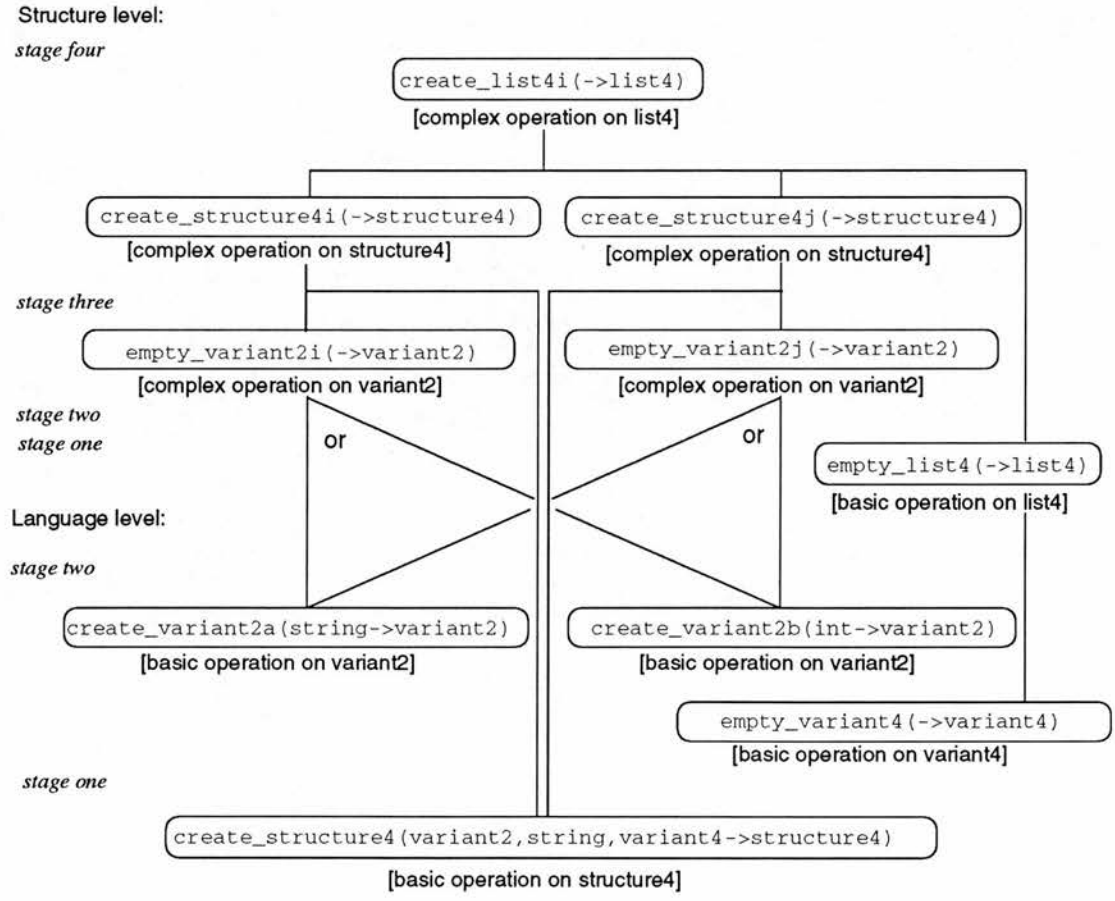


Figure 5–36: Illustration of operations related to `list4`.

level analysis for the rule base including the operation to create it, and thus, illustrate how the Symbol Level description was built.

The Knowledge Level description indicated that a program playing the game should contain knowledge of the rules of the game (figure 8–3). From building the program I believed that those rules were implemented in a rule base and that the data structure `list5` was the type of the rule base. Thus, my initial interpretation was that `list5` represented the type of the rule base which contains the rules of the game. This interpretation was checked by inspecting the structure of `list5` and placing further interpretations on its subcomponents. Figure 5–33 shows the structure of `list5` obtained from the transformation process. Figure 5–37 shows how this structure was interpreted in terms of the rule base.

The rule base is constituted of rules where each rule is represented by `structure5`. In the left-hand side of each rule there are three kinds of elements: two sets of patterns, represented by `list3` and `list4`, and a test (`variant7`). All patterns in a set are of the same kind: pattern1 (`structure3`) or pattern2 (`structure4`). Pattern1 represents patterns without variables, and its fields contain strings. Pattern2 represents patterns that can contain variables (points of the board). This variation is represented by `variant2` where variable points are `string`, and item points are `integer`. The other element of pattern2 is a `string` item (colour of *perepere*). Once the structure of `list5` was completely interpreted in terms of the rule base (following it top-down as we have just seen), the checking was completed by following this tree in a bottom-up fashion. In this way, I checked that the interpretations on subcomponents of `list5` fit well once they were all put together.

Interpreting `list5` as the rule base also requires its operations to be interpreted as operations on the rule base. This is illustrated in figure 5–38 with the operation to create the rule base (following from the example in figure 5–34). From looking at all the operations identified for `list5`, `create_list5` was interpreted as the operation to create the rule base. This initial interpretation was checked by following (in a top-down manner) the graph of relations between operations and their corresponding user-defined types. In this case, the rule

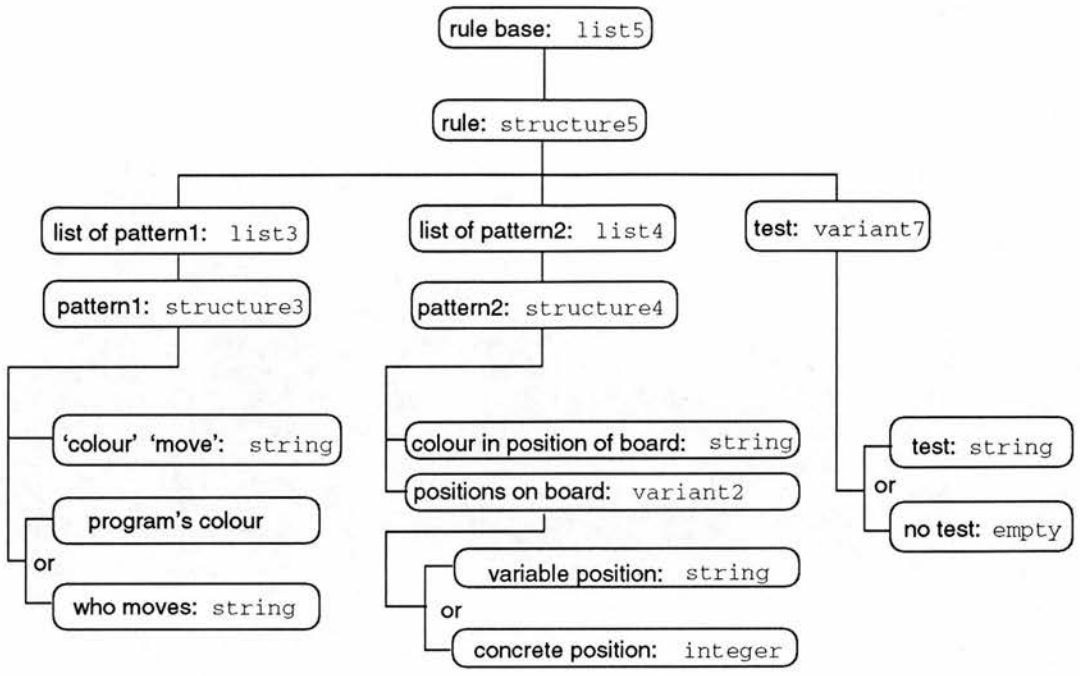


Figure 5–37: Illustration of the interpretations placed on user-defined types during the analysis at the domain level.

base is constituted of distinct rules which also had to be created. Having specific creation operations for each rule (`create_structure5i`) made it easier to interpret each as creating a distinct rule. There were specific creation operations for each set of patterns (`create_list3i`, `create_list4i`) and the test (`create_variant7i`) in each rule.

This top-down process continued until the operations identified at the language level were reached. Once operations at the language level were interpreted, the checking was completed by following the graph again, this time in a bottom-up way, and by confirming that the initial interpretations fit well with the functionalities of the operations (identified from the transformation process). For example, in figure 5–38 we can see how `create_variant7i`, which is specific to rule i, uses one of the two operations identified at the language level on `variant7`: `create_variant7a` or `empty_variant7`. These two operations were applied across rules depending on whether a rule involved a test, `create_variant7a`, or not, `empty_variant7`. At the end of this process the interpretation that `list5` represented the rule base was confirmed. This interpretation, however, was more

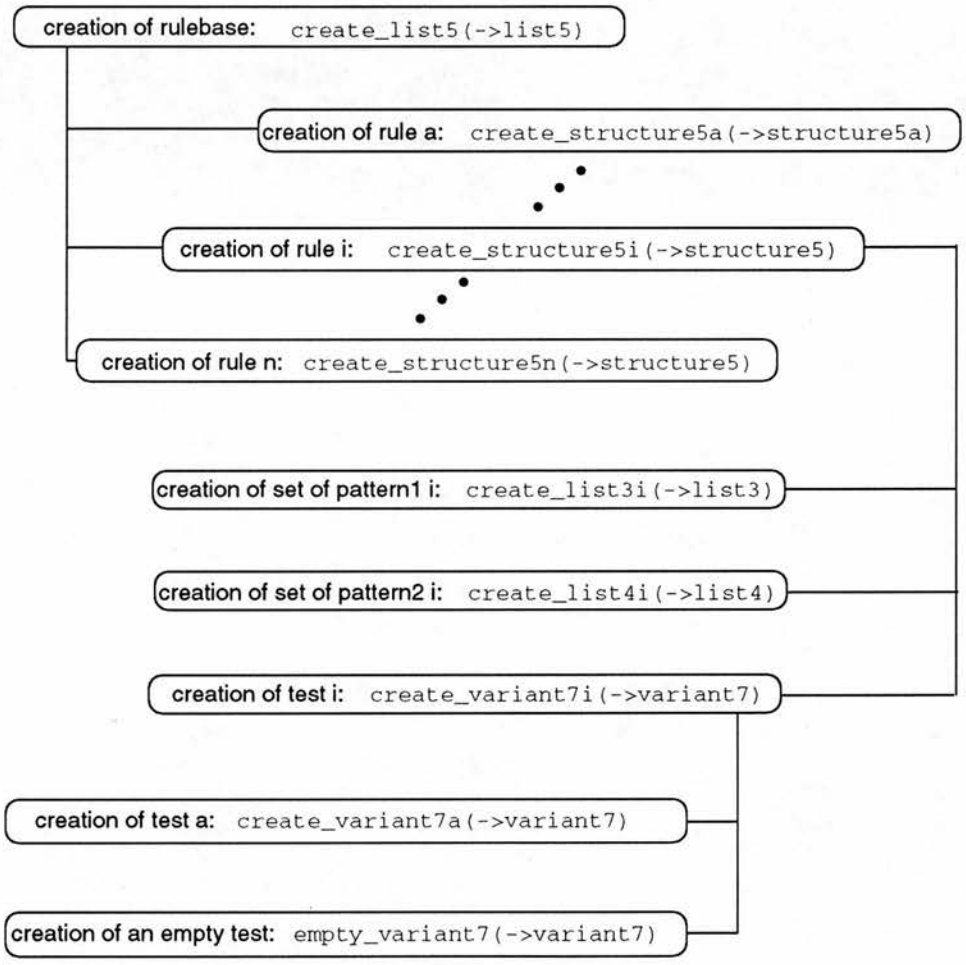


Figure 5–38: Illustration of the interpretations placed on operations during the analysis at the domain level.

detailed and clear in terms of what the rule base contained. The rule base did not contain rules for all the rules of the game, and some of its rules represented something other than rules of the game. For example, there were six rules representing the three possible forms of move (rule 4 of the game), one for each move and colour, and there was no rule representing rule 5 of the game, instead it was implicit in the way the board is built (represented by `list1` where each position only can have one colour). Thus, at the end of the domain level analysis, the rule base, as described in figure 5–37, i.e., with the structure of dependencies that constitutes it, and with a description of what each rule represents forms part of the Symbol Level description. This is also the case of all its operations. For example, the description of the creation of the rule base in figure 5–38 is part of the Symbol Level description. The Symbol Level also contains descriptions of the fact base (which includes the representation of the board), the working memory, and the agenda (represented by `list12`, figure 5–32), as well as their operations.

5.3.6 Analysis of Prototype Two

In this part of the experiment I investigated the possibility of using the results of the analysis on prototype one in the analysis of prototype two. This involved investigating a way of modifying the analysis method so that it coherently incorporates results from the previous analysis into the analysis of the new prototype. The analysis of prototype two began in the same way as before: division into modules, transformation process, and domain level analysis.

No new modules were identified in prototype two, so the program was divided into the same nine modules as prototype one. There were no new user-defined types identified, although new or modified operations were identified. Transformations were performed at the same levels and the same number of stages as prototype one, but there was a modification to the steps followed at each stage. A new step was added to the previous four: the initial step where, for each module, unchanged parts were replaced by their transformed forms from prototype one (see figure 5–39). The resulting module was the initial module for the analysis. In modules that did not change the initial step was the only step

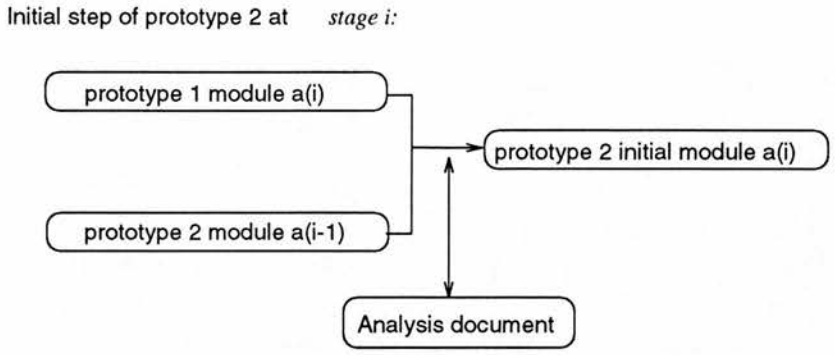


Figure 5–39: Initial step added to the method for the analysis of incrementally built prototypes.

performed at each stage (e.g., in types and printing modules). In modules that changed from prototype one (e.g., the rule base module where strategic rules had been added) the five steps were performed at each stage of each level.

During the domain level analysis, results from the domain level analysis of prototype one were again used. In this case, interpretations were placed only on the newly identified or modified operations and user-defined types, as the interpretations on the old ones were already placed in the analysis of prototype one. For example, in the rule base module, interpretations were placed on all the strategic rules added to the rule base, but not on the rules that already existed. The interpretation on the new rule base was also changed: it was interpreted as containing the previous rules (like in prototype one) plus the strategic rules.

5.3.7 Results from the Experiment

Abstraction Constructs. We have seen how persistence was successfully applied to provide information hiding (hiding implementational details from definition and use, and hiding analysed parts of the program from those that were still to be analysed). The dynamic and local nature of the persistent store was also successfully used to control the problem of managing the complexity of the analysis process.

Strong typing was successfully applied to provide increased precision for the interpretations placed by the researcher. It supported the definition of user-

defined types and operations, and its type checking system checked the reliability of the definitions and the subsequent transformations performed.

Analysis Method. The result of investigating further the initial scheme developed in the maze experiment was a method that:

- involves structuring a program where distinct functional parts are put in separate modules to be analysed separately;

- involves a transformation process at two levels of abstraction (language level and structure level) and that is aimed at the identification of user-defined types and operations by a process of program transformation. It is supported by the use of abstraction constructs. The identification step is based on code inspection. In the persistence step the identified user-defined types and operations are hidden in the persistent store. In the substitution step the program is transformed. Finally, in the validation step the transformed program is validated using type checking and a test set;

- involves a domain level analysis where the user-defined types and operations which constitute the new form of the program are interpreted in terms of the problem domain and the problem solving method used to provide a Symbol Level description;

- incorporates the analysis of incrementally built prototypes by adding to the basic method an initial step in which results from the analysis of previous prototypes are used without performing the same transformations again;

- incorporates the construction of an analysis document where relevant information on what is identified, and the transformations performed is recorded.

An important result was that placing interpretations at the domain level is easier using the new form of the program resulting from the transformation

process than with the original form. For example, it is much easier to interpret the rule base as represented by the identified `list5` and its operations (see figures 5–33, 5–34, 5–37, 5–38) than having to go through the 17 structures and 19 variants deciding which ones have to do with the rule base and why while attempting a coherent interpretation. It is also much safer because the new form (e.g., `list5` and it operations, figures 5–33 and 5–34,) was tested during the transformations. Thus, the method has been shown to be effective in supporting the analysis of AI programs by transforming them to a form that is easier to understand on inspection.

5.3.8 Observations

Test Sets. Defining good test sets for each prototype was not easy. It required analysing the behaviour of the program in terms of what it was meant to do (play the *Mu Torere* game). The design of these test sets was based on the Knowledge Level description.

Renaming. It was observed that operations that at the language level were identified and made persistent with parameters of one type (say `variant1`) were used (from the persistent store) at the structure level with the type of the parameter renamed (say to `list1`). This renaming process did not give rise to errors such as the type system indicating that two different names were used for the same type. That is, the type system accepted both names as corresponding to the same user-defined type without any problem. I will discuss this important observation in section 5.3.10.

Modifications. I again encountered the problem of having to transform an assignment operation into a creation operation to avoid compilation errors, thus changing the style of programming from imperative to functional (this was discussed in section 5.2.7). Another transformation problem I found in prototype two was that it was difficult to analyse the Napier88's clause `project` (which allows a value to be injected into a variant to be rebound to a constant location,

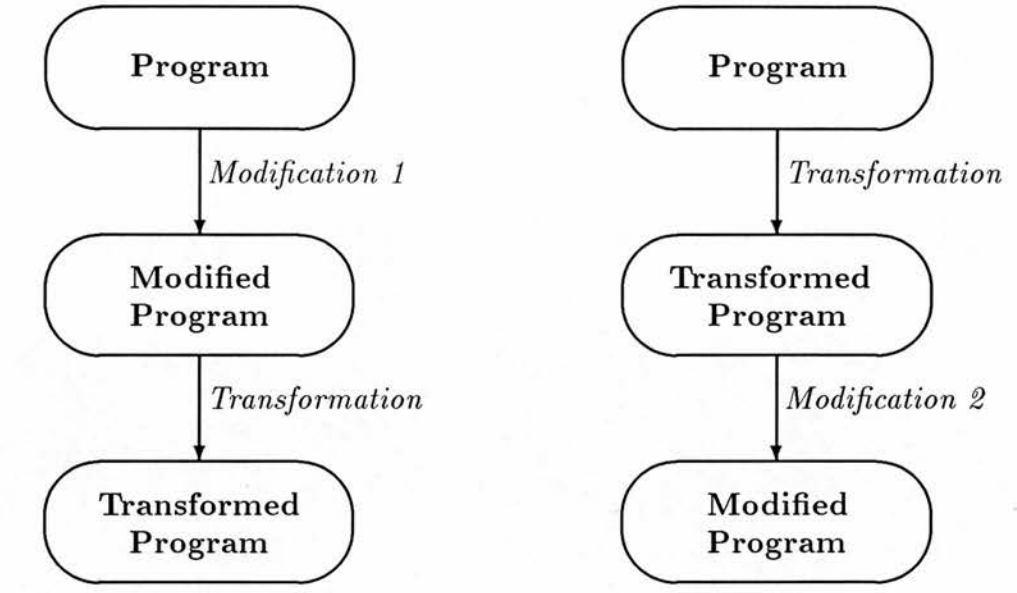


Figure 5–40: Two kinds of modifications.

thus facilitating static type checking). I found that this clause could not be analysed just as an operation on `variant` because operations on `structure` were also part of the clause. As a result, the neat distinction between structures and variants, and their operations, would not hold for this case. Two solutions were considered: to attempt the analysis of this special clause as it was, thus breaking the uniformity of the method, or transforming this clause to the form used in prototype one so that it could be analysed like the rest of the program. In this way, the uniformity of the method was maintained, and for this reason I chose the second solution.

These two kinds of problems involved two kinds of modifications: a *post-transformation* modification where the result of a transformation was transformed to a more functional form (instead of an imperative one) and a *pre-transformation* modification that involved modifying a language clause to a form that was easier to transform (see figure 5–40). In section 5.3.10 I will discuss the distinction between these two kinds of modifications and their relation to the transformations as part of the analysis method.

Incremental Analysis. The method helps to see how and where changes to parts of a program affect parts that had not been changed from the previous prototype. For example, new operations identified in the choice module affected, and thus forced changes in, the inference engine since it now used the changed choice operation. If the inference engine was not changed during the analysis, it would use the old persistent choice (from the analysis of prototype one) and the program's behaviour would be erroneous which would help to identify that the inference engine had to be changed to use the new persistent choice. Thus, the method helped the analysis of the second prototype so that a clearer understanding of how and where changes and additions to some modules affected others.

Documentation. Although the method provides the means to document the analysis in a structured way, documenting is still very complex. We have seen how not only the identification and substitution steps need to be documented but also all the changes resulting from errors during validation (see figure 5–31). Using the analysis document was not easy because there were no facilities for visualizing it. Thus, documenting the analysis, and making use of it, also adds to the complexity of the analysis. The support provided by the method for documentation will be discussed in section 5.3.10.

Domain Level Analysis. The analysis at the domain level was still a complex task (although it was made easier by the transformed form of the program as explained in section 5.3.7). The graphs of dependencies between user-defined types and between operations were complex, and following them was not easy. This task was made more difficult by the lack of an effective visualization of the program in the persistent store. This forced me to perform this analysis on the basis of the documentation, and not on the inspection of the program itself, which was in the persistent store.

5.3.9 Assessment

This experiment successfully achieved all its objectives. The Prototype Analysis Method was developed based on the scheme resulting from the maze experiment. The transformation process (supported by abstraction constructs and incorporating facilities for the analysis of incrementally built programs) and domain level analysis are two distinct parts of this method. A Symbol Level description is obtained at the domain level as a result of interpreting the user-defined types and operations identified during the transformation process. Obtaining this description is much easier in the new form of the program.

5.3.10 Discussion

On Test Sets. Defining suitable test sets is an important problem in any AI research experiment. The understanding of the kind and range of behaviour displayed by a program is provided by the dynamic analysis (explained in section 2.4.1) from which a test set is also obtained. By using test sets from the dynamic analysis for validation, the method incorporates an important link between the dynamic and static aspects of the analysis of a program.

Renaming and Structural Equivalence. A question arising from the renaming process (section 5.3.8) is: how does the type system know which user-defined type name is correct? The answer lies in the kind of type equivalence rule used in the Napier88 type system—structural equivalence. Structural equivalence allows the use of different names for the same type so long as the structure of the type remains the same. This is very important in the analysis. Without it, the operations in the persistent store would need to be changed each time a user-defined type was renamed, and this would add a lot of complexity to the analysis. With the use of structural equivalence there is no need to change names in the persistent store each time a new name is given to a user-defined type. Thus, managing the renaming process does not require any extra effort.

On Modifications. The two kinds of modifications described in section 5.3.8 (pre-transformation and post-transformation modifications) were performed, not to identify user-defined types and operations, but to allow the analysis to be performed uniformly. It is important to analyse the modifications performed during the analysis (which are different from the transformations) to understand their relation to the application of the method. The two kinds of modifications described in section 5.3.8 were due, in a strong way, to the language used for the experiments. If a functional language had been used, instead of an imperative one, the post-transformation modification might not have been necessary. The same applies to the `project` problem, as the problem would not have occurred in a language without that particular clause. However, other languages are likely to produce other problems, and their solutions may also involve specialised modifications.

On the Method. The method provides a good support for the analysis of programs, particularly if they are built by incremental prototyping. It does not, however, provide good support to manage the complexity of adding and viewing information in the analysis document. The support that persistence provides for complexity management does not include supporting the visualization of its content. This is a drawback of the particular version of Napier88 used. More recent developments of the system include an adaptive graphical browser, [Kirby & Dearle 90], to support this visualization.

The complexity of the domain level analysis (see section 5.3.8) is directly related to the method's lack of support for documentation and the lack of support for the visualization of the persistent store. Better support of these issues will result in the method providing better support for the domain level analysis.

For the analysis of further prototypes, the method can be improved by adding an intermediate validation step after the initial step. In this way, errors from the initial step are separated from errors in the other steps.

Other ways of improving the method involve finding ways of increasing the reliability checking. One way of adding further support in this area could be by

defining the semantics of the operations identified. This would provide a better definition of the operations, and thus, increased precision.

5.3.11 Outcome of the Mu Torere Experiment

The outcome of this experiment was a method to support the static analysis of experimental AI programs, the Prototype Analysis Method. Persistence, strong typing, and structural equivalence are abstraction constructs successfully and effectively incorporated in this method to support its application, and thus, the analysis. The method is based on levels of abstraction (at which program transformations are performed) and on test sets (from the dynamic analysis) for validation. The analysis of incrementally built prototypes is made easier by using results from the analysis of previous prototypes. The researcher's incremental understanding (during the transformation process) results in a new form of the program that is interpreted at the domain level, and from which a Symbol Level description can be more easily constructed.

5.4 Summary

In this chapter I have presented the experiments performed to investigate the use of abstraction constructs to support the analysis of AI research programs. The outcome of the stack experiment was an initial identification of the role played by Software Engineering abstraction constructs in the analysis (except polymorphism) and the identification of three levels of abstraction. From the maze experiment an initial scheme to perform the analysis was identified. Its application was supported by the use of persistence and strong typing but not abstract data types. Finally, in the *Mu Torere* experiment a complete analysis method was developed for the analysis of incrementally built AI research programs—the Prototype Analysis Method—whose application is supported by the use of persistence, strong typing, and structural equivalence.

Chapter 6

The Prototype Analysis Method

6.1 General Description of PAM

The Prototype Analysis Method is aimed at helping a researcher perform the static analysis of an experimental AI program built from weak (and possibly changing) specifications, and by incremental prototyping. The method does not rely on specifications or clear documentation. This kind of analysis is not a process of verification, and thus, the method does not aim to check whether a program satisfies a given specification (as is the case in some of the other research on program understanding that we have seen in section 3.6). Instead, it relies on the program code, and the researcher's ideas of what he or she attempted to do in building the program. These ideas may or may not be documented, and it is often the case that they reside only in the researcher's head.

The method involves separating the program into modules, a transformation process, and a domain level analysis. During the transformation process, a program is transformed at two levels of abstraction: language level and structure level. The transformation process helps a researcher to transform a program to a form where data structures and operations are more clearly identified, abstracted from implementational details. This new form is easier to understand on inspection at the domain level, where interpretations are placed on the identified data structures and their operations. These interpretations relate the data structures and operations in their new form to the ontology of the problem domain or problem solving method used.

PAM also takes into account the incremental nature of the building process by helping the researcher in the analysis of each incremental prototype program built on the way to the final system. Each prototype is analysed separately and in the sequence of their development. The method incorporates facilities so that results from the analysis of a prototype are used in the analysis of subsequent prototypes, and they add to the researcher's understanding of the system. Practically, this can also reduce the amount of code to be analysed in further prototypes. This is particularly the case when parts of a program don't change from one prototype to the next. For these parts, the method incorporates a step where the analysis performed on a previous prototype is used in the actual prototype, instead of repeating the analysis.

Dividing a Program into Modules. During the analysis of a program it is important to record where in the program data structures and operations are found, as it helps to understand better their representational role (i.e., a particular type of data structure may represent different things in different contexts). To facilitate this task a program is divided into modules at the beginning of the analysis. Modules are intended to contain separable functionalities or contexts within a program. (See section 5.3.4 for an example.)

6.1.1 The Program Transformation Process

The aim of the program transformation process is to abstract away implementational detail to reveal the program's essential causal and representational structure. This is done by identifying, at a series of levels, user-defined types and operations on them. This process of abstraction is also a process of *specialization* (in contrast to generalization) because to understand the representational role of instances found in a program it is necessary to know the particular user-defined type the instances belong to (and the operations on it).

User-defined types specified by the program builder describe patterns of data structures used in the program to represent particular aspects of the domain. User-defined types occur at two different levels of abstraction.

At the language level, user-defined types are formed from specializations of the constructs available in the language used. For example, a user-defined type might be formed from a particular type of list, in Lisp, or structure, in Napier88, and which is used to record the current positions of pieces in some board game. At the structure level, user-defined types are specializations of data structures that can be formed from a combination of user-defined types identified at the language level. However, sometimes user-defined types identified at the language level also occur as user-defined types at the structure level. This can happen when, in the language, there is a construct of a type that can be directly specialized to form a user-defined type at the structure level. For example, a user-defined type of type list might be formed by a combination of user-defined types, identified at the language level, formed from specializations of language constructs `variant` and `structure`, in Napier88. However, in Lisp, this user-defined type could be formed directly by specializing the list construct.

To uncover a program's essential causal structure, we need to have a definition of the operations applied to the user-defined types. Thus, during the transformation process, we seek to obtain a description of all the user-defined types and their operations at the structure level. However, getting such a description straight from a program is very difficult. It is therefore easier to start by obtaining a description at the language level where user-defined types and operations are also identified. Their form will depend upon the constructs available in the language and the way they are used to implement the program. This form offers a more uniform description of the data structures used by the program and of the computational operations performed on them. From this form, a structure level description can be obtained more easily.

Each level consists of a number of stages; each stage involves four steps: identification, persistent object definition, substitution, and validation. These steps are repeated to form as many stages as are required to complete the program transformation at each level. This two-level multi-stage transformation process is shown in figure 6-1 and will now be described in more detail, starting with the language level. During this description I will point to a small example in appendix A of the process and transformations performed on a print operation during the application of the transformation process. To show the kinds of elements identified at each level, I will provide some simple examples. The examples are provided in both Napier88 and Lisp [Wilensky 86] to show the more general applicability of the method, independently of the particular language used for the experiments.

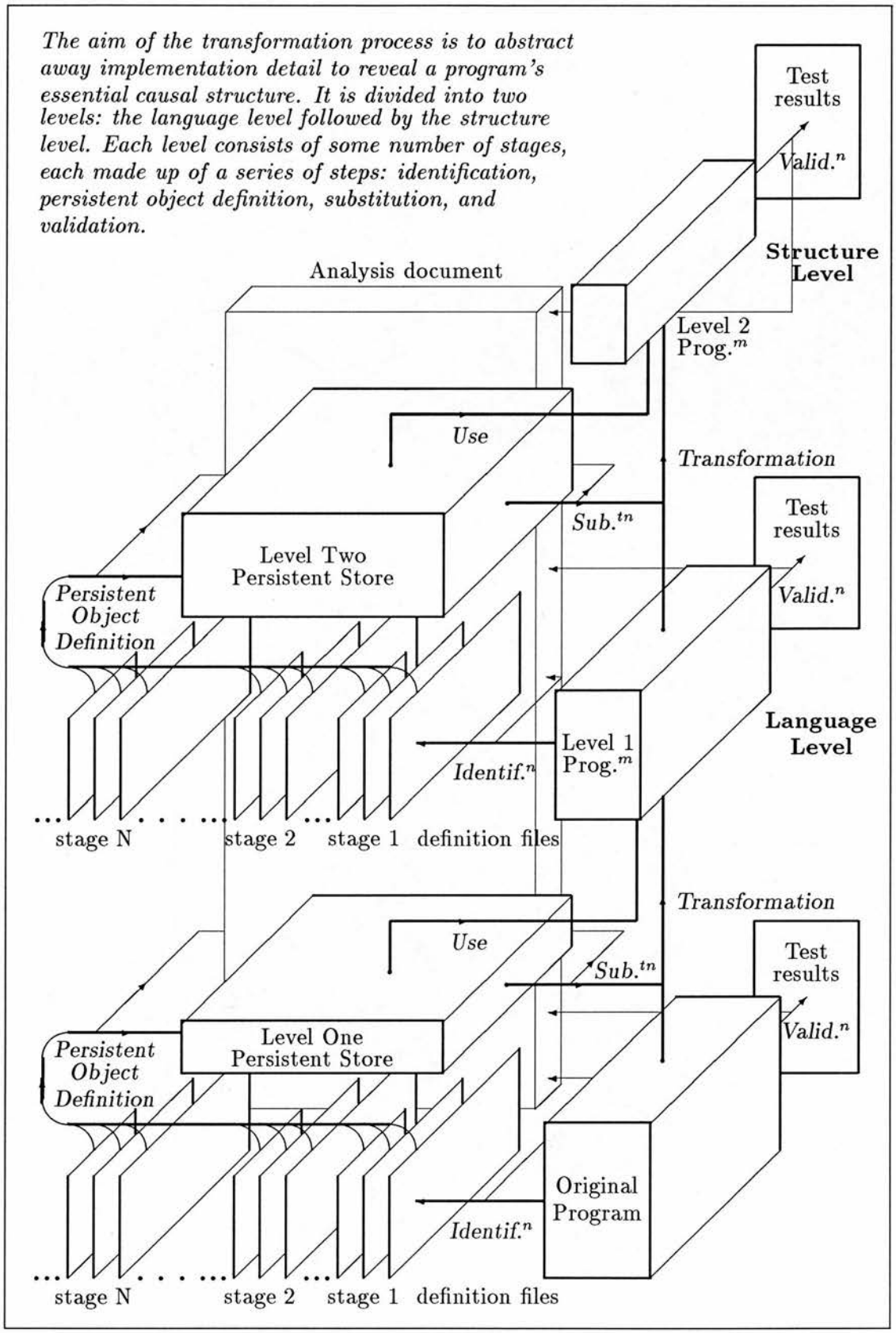


Figure 6–1: Program Transformation Process.

The Language Level

At the language level user-defined types and the operations identified are specializations of the general (basic) types and their operations available in the implementation language. They are also abstractions of their particular instances present in the program. They are identified by using the type definitions in the program. This is done by a process of induction from objects and operations in the original program (to know where they are used). The aim of the analysis at the language level is to create a version of the program in which the user-defined types and operations are uniformly identified and typed and which are also made persistent. This aim is the same for programs implemented in languages like Napier88 and languages like Lisp.

Abstracting from instances. Each user-defined type identifies a different specialization of the same language type and expresses all instances of the form it describes. By choosing one form for these instances, in which to re-express all instances, greater uniformity can be introduced into the program, making it easier to identify necessary variations as opposed to arbitrary ones arising from implementation choices. For example, in a Napier88 program we might identify a user-defined type `struct1` (a specialization of the Napier88 type `structure`) as having two fields, `a` and `b`, where `a` is of type `integer`, and `b` is of type `string`, after finding one or more instances of structures of this type in the original program (and the type declaration with another name). Alternatively, in a Lisp program, we might identify a user-defined type `list1` (a specialization of the type `list`) or `association_list1` (a specialization of the type `association list`) or `struct1` (a specialization of the type `structure`).

Each type in a programming language also has associated with it a set of operations: creation, access, modification, etc. In Napier88 the type `structure` has indexing operations that take the form <name of instance>(<name of field>). In the previous `struct1`, indexing operations of the language type `structure` will appear as `s1(a)`, and `s2(b)` (where `s1` and `s2` are instances of `struct1`). Alternatively in Lisp, there are operations associated with lists like `(car l1)` which returns the value of the first element of the instance `l1` of `list1`. Association lists have operations such as `assoc`, e.g., `(assoc 'a al1)` searches the association list `al1` sequentially and returns the first element whose first argument

is a. Similarly, structures defined with `defstruct` have accessing operations like `(struct1-a s1)` which returns the value in `a` from the instance `s1` of `struct1`.

The operations described above can all be seen as specializations of indexing operations. In the case of `struct1` two indexing operations might be identified in a program—`index1-a` and `index1-b`—where `index1-a` is defined as a function that takes an instance of type `struct1` and returns a value of type `integer`, and `index1-b` is defined as taking an instance of type `struct1` and returning a `string`. Specializations on the operations provide greater clarity over the operations used in a program. For example, `s1(a)` may also be an instance of a `struct2` that also has a field named `a`. By specializing operations over `struct1` separate from those over `struct2` (i.e., `index2-a`) greater clarity over the way each user-defined type is operated is introduced.

Abstracting from different languages. User-defined types also abstract over their implementation in different languages. For example, the definitions of indexing operations on `struct1` are abstracted from their implementation either in Lisp (using `defstruct`) or Napier88 (using `structure`):

`index1-a::  s:struct1 → integer`

where its body in Napier88 is `s(a)` and in Lisp `(struct1-a s)` and

`index1-b::  s:struct1 → string`

where its body in Napier88 is `s(b)` and in Lisp `(struct1-b s)`. As we can see, the particularities of the language are buried in the body of the operation, but the definition of the operation is the same. This can only be seen as an abstraction when the data type is present in the languages being compared. It cannot be seen when comparing Napier88 and Lisp in the case of `list1`, for example, because Napier88 does not offer the type `list` as a basic language type. Indexing operations of `list1` and `association_list1` in Lisp can be defined as follows. In the case of `list1`:

`index1-a::  l:list1 → integer`, the body is: `(car l)`; and for

`index1-b::  l:list1 → string`, the body is: `(cadr l)`.

In the case of `association_list1` the result of the operation `assoc` is a `list`. `Assoc` indexes over the name of the field, but it doesn't access the value of the field, it just returns the field. Thus for:

```
index1-a::  al1:association-list1 → list2
```

with body `(assoc 'a al1)`; and

```
index1-b::  al1:association-list1 → list3
```

with body `(assoc 'b al1)`

where `list2`, and `list3` are also identified as user-defined types (specializations of `list`). They might both be defined as having two elements: in `list2`, the symbol `a` and an `integer` associated with it (e.g., `(a 3)`); in `list3` the symbol `a` and a `string` associated with it (e.g., `(b "z")`).

Stages at the language level. The language level is divided into a series of stages. Each focuses on user-defined types that are specializations of one particular language type. For example, stage one might look for all the user-defined types that can be identified as specializations of the language type `structure` (`struct1`, `struct2`, ... ,`structN`) and their operations, stage two might look for `vector1`, `vector2`, ... ,`vectorN`, and their operations, etc. There are usually as many stages as there are different language types used in the program to form user-defined types.[1](See section A.2 for an example of two stages, one for the Napier88 type `structure`, section A.2.1, and another for the Napier88 type `variant`, section A.2.2).

Steps in each stage. Four steps are performed in each stage: identification, persistent object definition, substitution, and validation.

Identification: The identification step involves code inspection and induction to identify and specify user-defined types and their associated

---

[1]Note that basic types like `string`, `integer`, and `boolean` are not analysed. The analysis only considers types that can be specialized by user-defined types.

operations. Each user-defined type covers all instances of a language type specialization, and in this step its name and definition are obtained. For each operation its name, its body, the form of the call to the operation, and where in the program instances of the operation are found are identified. This information is recorded in the analysis document.

Persistent object definition: Once user-defined types and operations are identified and recorded, their definitions are separated from the program. That is, their definition is abstracted from their use, thus removing their implementational detail. In doing so, the parts that are already analysed are separated from the rest of the program which has not been analysed yet. This separation is achieved making the definitions of the identified user-defined types and operations persistent.

Substitution: In this step, the names of the newly identified user-defined types are incorporated into the program, so that they substitute their previous names. For example, the identified `struct1`, `struct2`, ... , `structN` substitute the language type `structure` (or the domain dependent name they could have in the program). Also, the occurrences of the operations applied to instances of each user-defined type are substituted by calls to their persistent definitions. For example, all the indexing operations on the field `a` of instances of `struct1` (e.g., `s1(a)`, `s2(a)`, `s3(a)`), are substituted by calls to the operation `index1-a` (in the persistent store) applied to `s1`, `s2`, and `s3`. See section A.2.1 for an example of this. This substitution is carried out using information in the analysis document about each user-defined type, each operation, and where in the program it is to be found. This way it is not necessary to scan the program again. Information on the patterns found for each operation and the substitution performed in this step is also recorded in the analysis document.

Validation: Once all the substitutions have been performed, it is necessary to check that definitions in the persistent store are syntactically correct, that their use is reliable, and that the program's behaviour has not been altered. The program is run using the persistent store, and a test set resulting from the evaluation of the behaviour and performance of the program during the dynamic

analysis. Once the changes are checked, and any necessary corrections made, the analysis proceeds with the next stage.

At the end of the language level analysis all the user-defined types and their operations are identified as specializations of language types and operations, and a uniform definition and use of the types and operations is obtained. For an example of the resulting program after the language level see section A.2.2.

The Structure Level

The transformation process continues with the next level of abstraction: the structure level. The transformed program together with the operations in the persistent store and the analysis document from the language level are the input to the structure level analysis (see figure 6–1). For example, at this level the aim is to establish what specializations of data structures (and associated operations) are employed in the program (which are built from the user-defined types and operations identified at the language level). At this level we should seek to understand questions like: does `list1` really represent a `list` or does it represent a `structure` (implemented as a `list`)? We should also seek to understand the relations between the different user-defined types. For example, whether `list2` is used to define only subelements of `association_list1` (as in the example) or also some other kind of element independent of `association_list1`.

Implementing a certain data structure in a programming language depends mainly on two things: the constructs provided by the language, and the choices by the program builder on how to use these constructs. Napier88 does not provide a type `list` as a data structure of the language but it can be implemented as a combination of two other language types: `structure` and `variant` (see section C.3 for how the combination is done, and section A.3.1 for an example). Alternatively in Lisp, a `structure` can be implemented using `defstruct` or as an `association list` or as a plain `list`. At the structure level then, a step forward is taken in understanding the program by abstracting the language dependencies and associated implementation choices made during the development of the

program. (See section A.3 for an example of the two stages of transformations at the structure level.)

Say that as a result of the language level analysis the user-defined type `association_list1` is defined as having two fields `a` and `b`, of type `integer` and `string` respectively, and that the user-defined operations found were `index1-a` and `index1-b`. From their definition (given earlier in this section) it seems likely that `association_list1` represents the same `structure` as the `struct1` described before. Say `association_list1` is interpreted as representing a `structure`, and more concretely as having the same definition as `struct1`. In this case the values of the fields `a` and `b` should be accessible using some kind of indexing or accessing operations. Nevertheless, from its operations we can see that the result of indexing a field in an association list is not the value of that field (unlike `struct1` in Napier88 or `defstruct` in Lisp). The result is the field itself (which, in the previous example, can be of type `list2` or `list3` depending on the field). Thus, in order to interpret `association_list1` as `struct1`, at the structure level new accessing operations need to be defined, say, `access1-a` and `access1-b`. I illustrate this with the definition of `access1-a`:

`access1-a:: al1:struct1 → integer`

its body is `(second2(index1-a al1))`, where `second2` is an operation identified at the language level on `list2`. Its definition could be: given an instance of `list2` it returns an `integer` that corresponds to the second element of that instance.

`second2:: l:list2 → integer`, its body is: `(cadr l)`

For this interpretation to be possible, it is necessary to have identified code in the program similar to the body of `access1-a`. If it is identified, and after substituting it with calls to `access1-a` the program behaves in the same way, this interpretation is confirmed. Otherwise this interpretation has to be checked on the basis of what is in the program.

If `association_list1` is interpreted as representing `struct1` its name is changed to that of `struct1` in the program, and the combination of `second` and `indexing` operations into calls to the previously defined accessing operations.

In this way operations performed on a user-defined type are added to its list of operations in the analysis document as the understanding of what each user-defined type represents is developed and established. Their definitions and bodies are added to the persistent store from where they are called in the program. As the transformation process progresses, code is removed from the program leaving a trace of definitions behind. When the program is run, the operations used in the program are type checked against their definitions in the persistent store. In this way, the reliability of the changes performed on the program (and the use of the persistent store) is validated.

From the example of `access1-a` it should be clear that more and more operations are defined during the transformation process that are combinations of operations identified at previous stages or the previous level. This happens mostly at the structure level where, from looking at the transformed code, more abstract operations that are combinations of operations on the same user-defined type can be identified (see section A.3 for an example).

This level finishes when there is no more code in the original file to analyse. At the end, and thus the end of the transformation process, what is left is a few calls (or maybe one) to persistent operations, and their definitions (as defined in the persistent store) (see section A.3.2 for an example). I illustrate the connection between the operations in the persistent store and their use in the program with the example of `access1-a` (see section C.7 for the use of persistence in Napier88):

```
access1-a = proc(s:struct1 -> integer)
 use ps with
  index1-a:proc(association_list1 -> list2);
  second2:proc(list2 -> integer) in
    second2(index1-a(s))
```

where `access1-a` is defined as a procedure that takes a parameter of type `struct1` and returns an `integer`. This integer is the associated value of the field `a` in an instance of `struct1` (which is implemented as `association_list1`). The implementation of `access1-a` involves the use of two already persistent

operations: second2, and index1-a. Index1-a indexes a in an instance of association_list1 and returns an instance of list2. This instance is the input to second2 which gets the second element from it: an integer and the result of access1-a.

The Persistent Store. At the end of the transformation process, the persistent store is not just a flat file of independent definitions. It is a structure of linked definitions. This structure represents the understanding of the program in terms of the user-defined data structures, relations between them, operations performed on them, and their implementations at the different levels. I illustrate the understanding obtained from the transformation process and how this understanding is reflected in the structure of the persistent store in figure 6–2.

In this figure we can see that one of the data structures appearing in the program is interpreted and defined as struct1 which is implemented as association_list1 (which is defined in terms of list2 and list3). Struct1 and its operations (access1-a) are identified, defined and made persistent at the structure level. For access1-a to be defined, index1-a and second2 need to be defined at the language level, and they are used from the persistent store to implement access1-a. Their definition in the body of access1-a is type checked against their definition in the persistent store, and if they are not the same an error will be produced.

Thus, we can see that the persistent store contains an explicit and type checked structure of operations and their connections. The dependency of one operation (access1-a in this case) on others (index1-a and second2) is made explicit; not just by calls to these operations as part of the body of access1-a (as it is the case when procedural abstraction is used) but by explicit definitional (typed) links. The same applies to the use of assoc in the implementation of index1-a, and car and cdr in second2. Thus, at the end of the transformation process, the persistent store contains a structure of three levels (for this example) of explicit and type checked connections which reflect the dependencies of operations and guarantee their correct use.

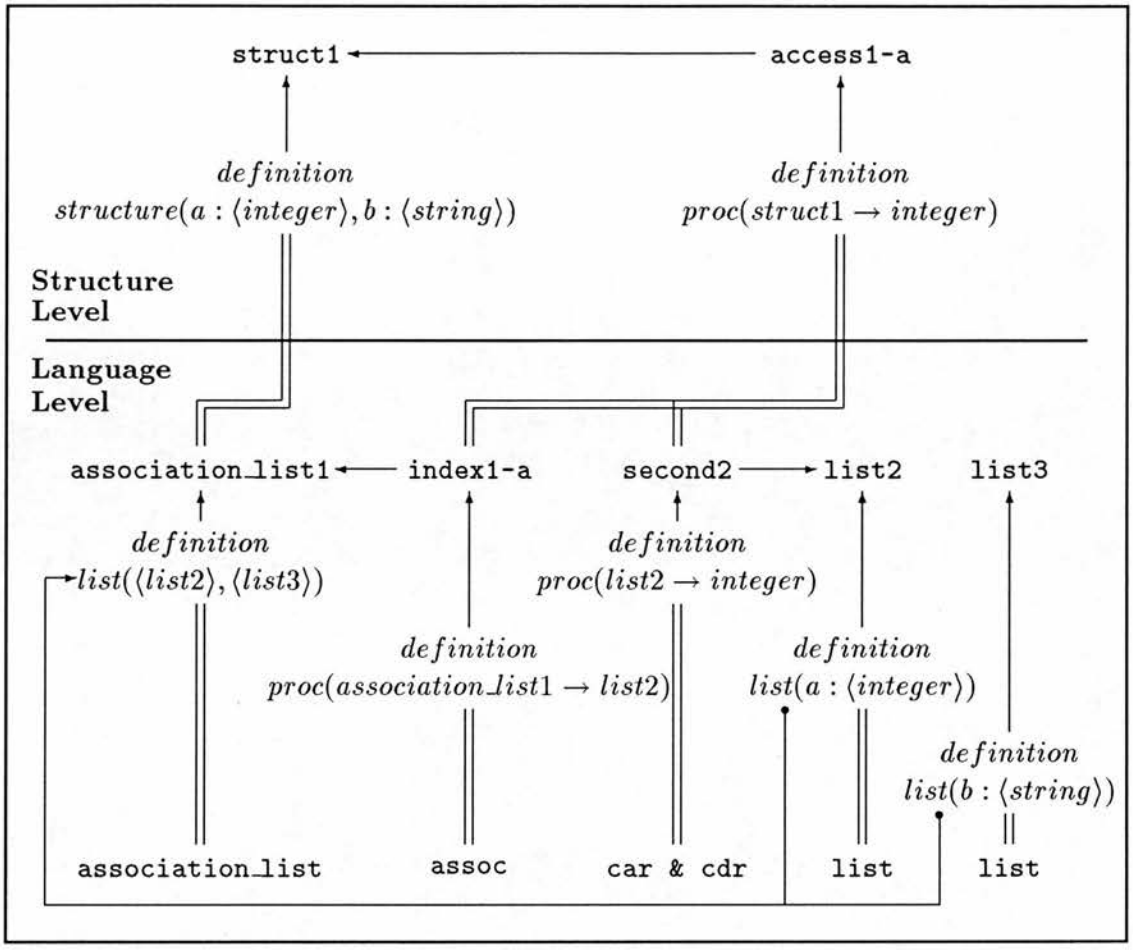


Figure 6–2: Example of a persistent store structure after program transformation. Double lines indicate implementation, and single lines indicate relation. For this example, at the end of the transformation process, the persistent store contains a structure of three levels of explicitly made and type checked dependencies.

At the End of the Transformation Process

The understanding and re-structuring of the program gained from the transformation process is kept in the persistent store and the analysis document. The program is re-structured, and large parts of it are hidden away as a graph of definitions and dependencies in the persistent store. The analysis document maintains a record of the re-structuring and the understanding gained in the process. This understanding and the re-structured program play a crucial role in the domain level analysis.

6.1.2 Domain Level Analysis

The analysis at the domain level is an attempt to understand what it is that the data structures and operations identified during the transformation process represent. Their representational role is interpreted in the context of the problem domain (e.g., the configuration of the game in a chess playing program) or the problem solving method used (e.g., a rule in a rule-based system).

The information used at this level is the program in the persistent store, the analysis document, the Knowledge Level description of the problem, and the knowledge of the program building experience. The interpretation is a top-down process (in contrast to the bottom-up approach of the transformation process): complex data structures are analysed before simpler ones. It is important to interpret a complex data structure first because it provides the context for interpreting its subcomponents. For example, in a rule-based chess playing program, the context provided by interpreting a data structure as the rule base can be used to understand a subcomponent of it as a rule of the rule base.

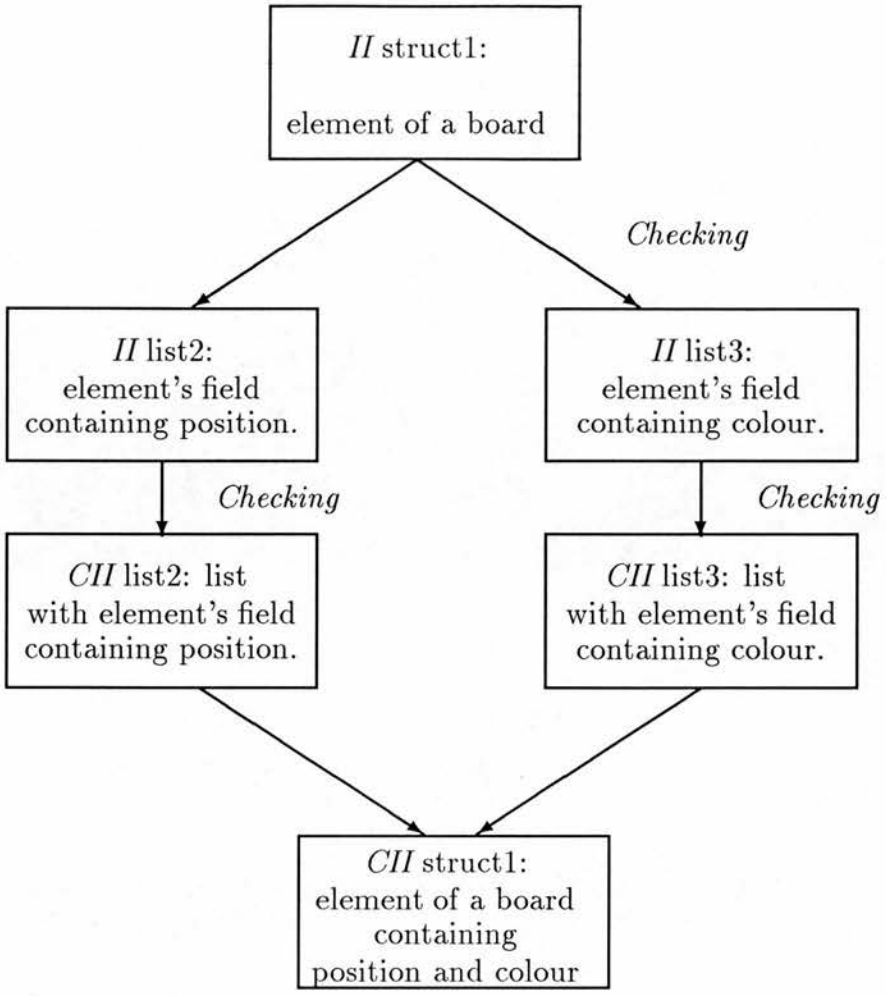
The interpretation of a data structure depends on its subcomponents and operations fitting into the attempted interpretation. Thus, the analysis of a data structure at the domain level is not complete until all its subcomponents and operations are analysed. The analysis is a cycle that involves initial interpretation and checking (from which the initial interpretation is confirmed or revised). The initial interpretation is performed on the basis of the Knowledge Level description and the researcher's ideas (and any notes or description) on the design. The checking is based on the information in the analysis document and the graph of dependencies in the persistent store.

An initial interpretation of a data structure is checked by analysing its definition and operations which may involve further cycles of initial interpretation and checking for each of its subcomponents. For example, analysing `struct1` involves providing an initial interpretation for it, and checking that interpretation by analysing its subcomponents `list2` and `list3` (each of which requires a further analysis cycle). It also involves analysing each of its operations, for which

the same analysis cycle applies, i.e., analysing `access1-a` involves providing an initial interpretation in the context of the interpretation given to `struct1`. That interpretation is then checked by analysing the subcomponents of the operation—`index1-a` and `second2`.

There are two important points about this level. One is the crucial role that the new form of the program plays in supporting the checking of initial interpretations of data structures and operations. The other is that at this level the method provides a way of performing the interpretations in an organised way, and in a way which attempts to minimize the effects of the researcher's preconceived (and possibly incorrect) ideas about the roles played by particular data structures and operations.

Figure 6–3 illustrates this process at the domain level for the example of the identified structure `struct1`. The domain for this example is taken from one of the experiments performed as part of this investigation, the *Mu Torere* game experiment (section 5.3). This game is played with a board that has nine positions (see figure 5–21). Each element of the board is identified by its position and the colour of the piece it contains. Initially `struct1` is interpreted as the type of each element of the board. For that to be true, it needs to contain a position (believed to be of type `integer`) and a colour (believed to be of type `string`). When this interpretation is checked against the findings from the transformation process, we can see that the implementation of `struct1`, `association_list1` contains two other data structures, `list2`, and `list3`. Thus, the cycle starts again, and initial interpretations are provided for `list2`, and `list3`. Following from the interpretation of `struct1` (which needs to contain a position `integer`, and a colour `string`) `list2` is interpreted as containing the position, and `list3` the colour. When this is checked, `list2` is found to have a key `a` and an `integer`, and `list3` contains a key `b` and an associated `string`. As these findings correspond to the initial interpretation of `list2` and `list3`, their initial interpretation is confirmed, and this second cycle finishes. The interpretation of `struct1` is also confirmed (it contains the elements that it is supposed to contain) and the first cycle also finishes.



II: Initial Interpretation
CII: Confirmed Initial Interpretation

Figure 6–3: Illustration of the domain level analysis.

Interpreting `struct1` as an element of the board is not finished after the data structure is checked. Interpretations over the operations performed on `struct1` are also performed to confirm this interpretation.

6.1.3 Analysis of Incrementally Built Prototypes

The transformation process and domain level analysis constitute the basic analysis method that can be applied to a program whether it is built incrementally or not. I have further developed the analysis method to incorporate features that can help in the analysis of incrementally built systems. Incremental prototyping

(as opposed to exploratory or experimental prototyping) involves the gradual building of a system via a series of functioning prototypes (see section 3.2.2 for a reminder of this). That is, each prototype is built by adding (or changing) something to the previous prototype. If prototypes are not built by incremental prototyping but by exploratory prototyping, for example, the basic method can be applied for each prototype.

This method takes advantage of the analysis performed on previous prototypes when analysing subsequent versions. Parts of a program that don't change can be substituted by their already analysed user-defined types or operations, thus avoiding re-analysing them. The analysis then only needs to concentrate on the parts that change or are added from one prototype to the next. The first prototype is analysed using the basic method. To analyse further prototypes, including the final system, in the transformation process a new step is added to each stage of each level—the *initial* step—which is performed before the identification step. In the initial step, parts of the program that remain the same from the previous prototype are substituted by the result of the analysis at the same stage in the previous prototype. These substitutions can be validated by performing an intermediate validation step after the initial step. The rest of the steps are performed on the changed or added parts. At the validation step, the transformed prototype (including old and new transformations) is validated. In this way, if a part that was thought not to have changed (and thus was substituted by the previous analysis) had changed, errors are likely to appear during validation. These errors can again help the researcher to understand misinterpretations or misunderstandings of the changes performed from one prototype to the next. The substitutions performed in the initial step are also recorded in the analysis document together with the information of the rest of the analysis on the new parts.

At the domain level, interpretations are placed on the new parts of the prototype. In a rule-based system, for example, where further prototypes add or change rules of the rule base, only the new or changed rules need to be interpreted at the domain level, thus, relying on the previous analysis for the unchanged rules.

In the next section I present the procedure for applying PAM. This procedure identifies, in more detail, the steps to be followed when analysing a program. The last section of this chapter describes the decisions to be made, and the information and criteria to be used when performing these steps.

6.2 Procedure for Applying PAM

1• Design the analysis document.

2• Divide the program into modules.

2.1• Check that the division works.

2.2• Decide in which order modules are to be analysed.

3◇ Perform the transformation process.

4□ Perform the domain level analysis.

3◇ Perform the transformation process.

For each level:

3.1• Identify initial set of user-defined types in the latest transformed version.

3.2• Divide level in stages.

3.3• Perform analysis at each stage.

3.3.1• Perform initial step.

3.3.1.1 *For each module:*

3.3.1.1.1• Compare module with same module from previous prototype.

3.3.1.1.2• Copy the transformed module from previous prototype.

3.3.1.1.3• Modify the transformed module.

3.3.1.2• Perform intermediate validation step.

3.3.2• Perform identification step.

For each module:

3.3.2.1• Identify user-defined types in module.

For each user-defined type in initial set (identified in step 3.1):

3.3.2.1.1• Look for user-defined type in the module.

If the user-defined type is in the module:

3.3.2.1.1.1• Rename user-defined type.

3.3.2.1.1.2• Make its definition explicit in the definition file.

3.3.2.1.1.3• If there is not a record for it in analysis document, create one.

3.3.2.2• Identify operations on user-defined types.

For each user-defined type present in the module:

3.3.2.2.1• Identify operations on user-defined type.

3.3.2.2.2• Define operations in definition file.

3.3.2.2.3• Record operations in analysis document.

3.3.3• Perform persistent object definition step.

- Create program with union of definition files: definition file.
- Make definitions persistent.
  - Compile and execute (or interpret) definition file.

If errors:

- Make changes to resolve them.
- If definitions are modified record them in analysis document.
- Re-attempt to make the definitions persistent.

If no errors, the definitions have been made persistent.

3.3.4• Perform substitution step.

For each module:

3.3.4.1• Copy module into another (transformed) module.

In transformed module:

3.3.4.2• Substitute old names of user-defined types for new names.

3.3.4.3• Substitute operations by calls to their persistent definitions.

3.3.4.4• Record substitution in analysis document.

3.3.5• Perform validation step.

- Form a transformed program using transformed modules.
- Validate transformed program.
  - Execute program using the test set from dynamic analysis.

If errors:

- Make changes to resolve them.
- If definitions are modified record them in analysis document.
- Re-attempt to validate the transformed program.

If no errors and the program's behaviour is preserved, the stage is finished.

4□ Perform the domain level analysis.

4.1• Get the list of user-defined types from the structure level.

4.2• Make the list explicit in the analysis document.

4.3• Get the list of operations.

For each user-defined type in the list:

4.3.1• Get the list of its operations.

4.3.2• Make the list of operations explicit in the analysis document.

4.4• Interpret user-defined types.

Until all user-defined types in the list are interpreted:

4.4.1• Assign initial interpretation to a (uninterpreted) user-defined type.

4.4.2• Check initial interpretation.

- Inspect definition of user-defined type.
- Interpret each user-defined type in the definition in a top-down way.
- Check interpreted definition in a bottom-up way.

4.4.3• Interpret operations on user-defined type.

For each operation:

4.4.3.1• Assign initial interpretation.

4.4.3.2• Check initial interpretation.

- Inspect body of operation.
- Interpret each operation in body in a top-down way.
- Check initial interpretation on the operation in a bottom-up way using its interpreted body.

6.3 PAM: Decisions, Information, Criteria

This section describes the decisions made, and the information and criteria used to make them during the application of the method. They also apply to further prototypes except when otherwise stated. The numbers in brackets in the titles correspond to the step numbers in the previous section.

6.3.1 Design the Analysis Document (1)

All the information that is collected during the analysis should be recorded in the analysis document: information of the data structures and operations found in the program (their definition, implementation, and use) and the transformations performed. This document helps a researcher in organising and performing the analysis (i.e., the information recorded during the identification step makes the substitution step easier), and in describing the understanding of the program resulting from the analysis. It should contain a set of records that are filled in during the transformation process, the domain level documentation, and miscellaneous things such as lists of user-defined types operations (steps 4.2 and 4.3.1.3) and notes considered of interest. I will show how this information is filled in, and the importance of recording it while describing the procedure in more detail.

The set of records are filled with information on the user-defined types identified during the transformation process (see appendix B for an example). For each user-defined type, its old and new name (fields **OLD NAME** and **NEW NAME**), the list of its operations, **OPERATIONS**, and information for each one, **INFORMATION OF OPERATION**, is recorded. This is to keep track of the understanding at different levels of the analysis. For each operation, **OPERATION** records its name and its old name in the program, if it had one, **BODY** its implementation, **CALL** the form of its call in the program (including the types of its parameters), **WHERE OPERATION** where in the program it is to be found, i.e., **MODULE** in which module, and **CHUNK** in which identifiable part of the module. There are as many **CHUNK** fields as places where the operation is found in a module. **PATTERNS OF BODY** records the pattern(s) of the instances found in a module, and thus, it is similar to the **BODY**. **SUBSTITUTION** describes the substitution of the pattern for the call to the operation, and **ARBITRARY DECISIONS** any decision made to facilitate the substitution, or make it more clear. Information is added to the records incrementally, at each step and at each stage. To make the document more manageable, it can be divided in parts, one per

stage. In this case, there may be more than one record for the same user-defined type.

The domain level part of the document does not contain (at present) a clear structure (at least not as clear as the records just described for the transformation process). It contains everything that is interpreted at this level, and in the order in which it is done. That is, each cycle of initial interpretation and checking is documented. Failed interpretations should also be kept. They are good records of any misunderstandings identified.

6.3.2 Divide the Program in Modules (2)

The division into modules makes the analysis easier as the effort is concentrated on a particular functionality at the time, instead of the whole program. The criterion for the division is to look for parts of the program that can be considered as separable functionalities. This step is trivial if the program has already been divided into modules during development. The number of modules or their size is not constrained. The information available for this division is mainly the researcher's knowledge about the problem being solved and the method used to solve it. Each module is usually put in a separate file.

The program's type definitions may be collected in one module, a types module, or they may be left in the part of the program where they were defined. Leaving them where they were defined helps to see what types are used in each module. However, types defined in one module can be used in others. Thus, during the identification step, every type has to be looked for in every module. Collecting them in one module is better because it makes the identification more uniform, i.e., all the types are available in the one place. In this case, the module of origin of the type should be noted to facilitate its identification (i.e., the type should be identified at least in the original module).

Check that the division works (2.1). This is performed by executing the program (as in the validation step). Errors in this checking may be produced by

moving code around. Examples of errors are code that is missing or duplication of definitions (which is detected by the type checking system).

Decide in which order modules are analysed (2.2). The order is decided by the analyser but it is best if the order is logical and easy to remember. This order is to be followed all the way through the analysis to help in performing it methodically. Thus, it is to be recorded in the analysis document.

For further prototypes. The division into modules of the previous prototype(s) should prevail as much as possible. However, new modules can be created to incorporate additions to the program, and previous modules can be changed to incorporate modifications.

6.3.3 Perform the Transformation Process (3)

Information at each level. At the beginning of the language level, the analysis document is empty, the original program is divided into modules, and the persistent store is empty. When analysing further prototypes, at the language level, the analysis document has the information from the previous analysis, the program is divided into modules (including new ones, if any), and the persistent store contains the form of the program from previous transformation processes. At the beginning of the structure level, the modules, the analysis document, and the persistent store are those resulting from the language level analysis.

Identify initial set of user-defined types (3.1)

Language level. User-defined types are identified by inspecting the type definitions in the program (which are in the types module). In this inspection the aim is to identify the specializations of the data types provided by the language that are used in the program, and that are defined by the different type definitions.

Structure level. User-defined types are identified by abstracting from the user-defined types identified at the language level, and that are listed in the analysis

document. Using the knowledge of the language, every user-defined type in the analysis document is inspected. From this inspection each user-defined type is identified as implementing a specialization of a data structure (e.g., tree, list, etc.) or a component of one. Some of the user-defined types from the language level will be direct specializations of a data structure. In other cases, a data structure may be implemented in terms of another (e.g., a structure in terms of a list), or it will be built out of a combination of others (perhaps because the language does not offer it as a primitive).

The list of user-defined types at each level contains the name and definition of each user-defined type and it is recorded in the analysis document. It provides a global view of what is to be analysed, and of how to organise the stages at each level.

For further prototypes. Only the user-defined types that are new in the program need to be identified. The others are the same as those from previous analysis.

Divide a level into stages (3.2)

Language level. This is to be divided in, at least, as many stages as language types are identified. The order in which language types are analysed (i.e., the order of the stages) is irrelevant.

Structure level. Simple user-defined types and operations should be analysed before complex ones because they are likely to be used to build the complex ones. At each stage, the analyser decides on a set of user-defined types and operations to analyse. The number of stages depends on the granularity of these sets, and it is not restricted. There can be as many as the analyser considers appropriate. More stages are needed if the sets are small (i.e., if only a few elements are analysed at each stage). Small sets are preferable to big ones because the analysis at each stage is more manageable. For example, a stage can concentrate on simple operations on a data structure (e.g., operations like

head and tail on all user-defined lists), another on complex operations of simple versions of a data structure (e.g., complex operations on simple lists), another on complex operations on complex versions (e.g., complex operations on complex lists) etc. Simple operations are those that define the data structure, e.g., head, tail, for lists. Sometimes, these may have been identified at the language level, depending on the data structure facilities of the language used. At other times, they will be identified at this level. Complex user-defined types are those whose definition involves other user-defined types (e.g., a list of integers is a simple list, whereas a list of different kinds of structures is complex). However, it should be noted that complexity is a relative term, i.e., some (complex) user-defined types and operations are simpler (or more complex) than others. For example, a user-defined type whose definition includes other complex user-defined types is more complex than its (complex) components. Therefore, the complexity of a user-defined type or operation should be measured in terms of the complexity of its definition and in relation to the complexity of the other user-defined types or operations.

For further prototypes. Each level is divided in, at least, as many stages as before. At each stage the same kind of elements are analysed (e.g., simple operations on a data structure). More stages may be necessary to deal with new user-defined and operations.

Perform Analysis at Each Stage (3.3)

The initial step is to be performed only when analysing further prototypes. The other steps are always performed. The set of user-defined types to be analysed at each stage is extracted from the initial set (see criterion above). Sometimes, one stage will involve several iterations of the steps below to solve validation errors or incorporate elements that were forgotten at the beginning.

Perform Initial Step (3.3.1)

Comparing modules (3.3.1.1.1). Each module is compared to the (functionally) same module in the previous prototype(s) by inspecting the code. From this comparison it has to be clear which modules remain the same, and for those that change, which parts change and which remain the same. This is valuable information for the rest of the steps since they are performed only in the parts that change. Therefore it is recorded in the analysis document. This comparison is performed at each stage because the understanding of what changes and what remains the same in the modules may change as the analysis progresses. For example, if an operation A uses an operation B, and B changes to B', then A needs to be redefined to incorporate B'. Unnoticed changes are reflected by errors in the validation step. These errors are a source of information for the next comparison.

Copy the transformed module (3.3.1.1.2). The transformed module from the previous prototype at the same stage is copied to the (initial) transformed module for the stage. If the module did not exist in the previous prototype, the transformed module is the module itself.

Modify the transformed module (3.3.1.1.3). If the module is new no modifications are performed. If the module has not changed from the previous prototype, the initial step is finished. If parts of the module have changed, the changed parts are incorporated into the transformed module. These changed parts are the result of the previous stage in the current analysis.

Perform intermediate validation step (3.3.1.2). This is performed to validate the understanding from the comparison. In this way, errors from the initial step are separated from errors in the other steps.

Perform Identification Step (3.3.2)

In the first stage of the language level, the modules are the original ones. Otherwise, they are the transformed modules from the previous stage. At the beginning of the structure level, the previous stage is the last of the language level. When analysing further prototypes, the identification is only performed on the parts of modules that change or on modules that are new.

Look for user-defined type in module (3.3.2.1.1)

Language level. Any item of information extracted from a user-defined type definition can help to identify the use of the user-defined type in a module, e.g., its name, or the type of its subcomponents. However, individual items of information may not identify the user-defined type uniquely (except for its name) because they may be used in more than one type definition. A record should be kept of which items are repeated and in which user-defined types the repetition occurs. Items of information on a user-defined type `structure`, for example, are the name of the structure, and the names and types of its fields. It may be that its name is the only thing that identifies the user-defined type uniquely but that it is not explicit in the module. If any other item is identified but it does not identify the user-defined type uniquely (e.g., a name may be used to name fields in different structures) the context in which the item is used needs to be inspected to reach a decision. Names of variables can be useful in this inspection but, when inspecting them, their use should be considered carefully: a variable name can be used for different types in different contexts.

Structure level. It may be that a user-defined type identified at the language level is also a user-defined type at the structure level, i.e., there is a data structure in the language implementing the type of the data structure being analysed at the structure level. In this case, there will be a record for the user-defined type (at the structure level) already in the analysis document because it will have been created during the language level analysis. This record indicates if the user-defined type is used in the module. If a data structure being analysed at the

structure level is not provided by the language, it will have been implemented using another language type or a combination of language types. In this case, a user-defined type (at the structure level) is in the module if the user-defined types that implement it (at the language level) are in the module. Thus, their record in the analysis document should be looked at to see if they are used in the module.

Rename the user-defined type (3.3.2.1.1.1)

Language level. The name in the program is changed to one that indicates which specialization it is and of which language type.

Structure level. The name from the language level is changed to one that indicates which specialization it is and of which data structure. Renaming is not necessary when the user-defined type is the same at both levels.

Make its definition explicit in the definition file (3.3.2.1.1.2). The definition file contains the user-defined types and operations identified in each module and stage. Although this means that information may be repeated in different modules, it provides a historic account of what was identified in each module and at each stage. Thus, the declaration of the user-defined type identified in the module is made explicit in this file (with its new name if it has one).

Record it in the analysis document (3.3.2.1.1.3). A record is created for each user-defined type that is analysed for the first time at the stage. The fields NEW NAME and OLD NAME are filled in at this point.

Identify Operations on the User-defined Type (3.3.2.2.1)

Language level. Operations are specializations of language operations on each user-defined type. A user-defined type is a specialization of a language type, thus, all operations provided by the language for a type are possible operations on the

user-defined type that specializes that type. These operations are identified from the programming language manual.

There are two ways of identifying if an operation operates on a user-defined type. One is to look sequentially for all possible operations of a language type, and to identify, for each, if it corresponds to the user-defined type. This is likely to involve a lot of fruitless search: not all the operations provided by the language will be used for each user-defined type. The other is to identify, for each user-defined type in the module, its associated operations. This is a better approach since the search is constrained by the user-defined type. Which user-defined types are in the module can be easily determined by looking in the analysis document or in the module's definition file.

Identifying an operation as operating on a given user-defined type is done by inspection of the code and induction from instances. This inspection is guided by information about the operation and the user-defined type. The information about the operation is obtained from the programming language manual. It includes the name of the operation, its syntax, and the type of its parameters and result. The information about the user-defined type is obtained from the analysis document and the definition file. It includes its old name, the types of its subcomponents, etc. Names of variables (representing instances of the user-defined type) can also be useful when inspecting a tight context.

For example, in user-defined types with named fields (user-defined types of type structure, for example) the names of the fields help to identify indexing operations on them. However, if names of fields are not unique across user-defined types (i.e., a name names a field in more than one user-defined type) more information is needed. This information can be provided by type of the field if the type is different in different user-defined types. It should be noted that, in the renaming process, only the name of user-defined types is changed, not the names of their fields (if they have any). Even if their fields are renamed (so that each user-defined type has distinctive field names) the program still contains the old (repeated) names. Therefore, the problem of identification of operations is the same at the language level. Not renaming the fields is preferable,

otherwise the amount of old and new names that need to be maintained can increase considerably. Not renaming the fields may seem problematic because the confusion of which user-defined type a field name corresponds to is carried further into the structure level, and thus, the difficulty of identifying which user-defined type an operation operates on remains. In fact, this is not to be a serious difficulty because, at the structure level the information gathered during the language level is used to identify operations on user-defined types. This information is kept in **OPERATIONS** and **WHERE OPERATION** for each operation, and it keeps track of which operations are identified for each user-defined type and where in the program they are found.

Structure level. Operations are identified by looking, in the analysis document, for all the previously identified operations on the user-defined type in this module. At this level, new operations are built out of existing operations. That is, an operation is identified by finding a minimal combination of language level operations and/or previously identified structure level operations that, in combination, define a new operation. What new compound operations can be found is not defined *a priori*. The identification is a process of exploration and discovery, not of recognition of known patterns. This process is guided by the analyser's knowledge of the task and how it was programmed. The analyser has to be open to the identification of possible operations that do not match preconceived ideas.

The call to an already identified operation is looked for in the chunk indicated in the record. There, inspection is necessary to see if a more complex operation on the user-defined type can be identified. If it can, it is important to make sure that all the operations in its body are calls to persistent operations. If they are not, it means that something else needs to be analysed before this operation can be defined (i.e., more intermediate stages are likely to be necessary before this one can be done). This uncovers relations between operations (and thus between their user-defined types) that may not have been recognised before.

Sometimes it is not clear which user-defined type an operation belongs to because its functionality (its role in the causal structure of the program) can be interpreted in different ways. If the functionality

is not sufficient to make a decision, the following criteria can be useful. When an operation only involves one user-defined type, it corresponds to that one. When it involves more than one, i.e., it has more than one argument, the operation belongs to the user-defined type it modifies, and thus returns as a result. If an operation returns more than one result (or returns one result but also modifies global variables) it is likely to implement more than one functionality, and thus, it should be divided into as many operations as functionalities it implements, each returning one result.

Define operations in the definition file (3.3.2.2.2). Each operation is defined in the definition file with a name that relates it to its functionality and the user-defined type it belongs to. Procedures (or functions) are used to define operations. If the operation was also identified in another module for the same user-defined type, both definitions should be identical in the definition files.

Record operations in the analysis document (3.3.2.2.3). If an operation is new, its name is written in OPERATIONS. In OPERATION its name is recorded (and the old one if it had one), BODY is filled using the definition in the definition file, and CALL is filled using the name of operation and the type of its arguments. Always, regardless of if it is new or not, WHERE OPERATION is filled in: the name of the module in MODULE, and in CHUNK(s) the smallest identifiable chunk(s) of code in which it has been found (e.g., the name of a declared procedure the operation is part of).

Perform Persistent Object Definition step (3.3.3)

To make the definitions persistent the union of the definitions in the definition files of each module are put together in a program. Tools like SCCS, RCS, and MAKE can help to obtain their union. This program contains the definitions and the necessary instructions to make them persistent.

For further prototypes. Definition files will contain all the user-defined types and operations identified in the new or changed parts of the program. Some of them will be new with respect to previous prototypes. Others will have been identified in previous prototypes but they are also used in the new or modified parts of the current prototype. Only the definitions that are new should be made persistent since the others are already persistent from previous analyses.

Errors. If there are errors while compiling and/or running the definition file, changes will need to be made in different places depending on the source of the error. The error message usually helps to understand what changes are required. If definitions are modified to resolve errors, the analysis document has to be modified to record the changes. There are three main kinds of error. A definition might be erroneous. In this case, the problem is in the definition file. A definition might already be persistent. In this case, the problem might be in the definition file (e.g., an operation has been given the same name as another operation already persistent). Resolving it requires modifying or removing a certain definition from it. The problem might also be in the persistent store (e.g., the persistent store still contains the old version of an operation that has been modified and made persistent again). Resolving it requires making local changes to the persistent store. Finally, an element invoked from the persistent store is not present. Maybe one of the standard elements of the persistent store, or a persistent operation being used, is not declared in the definition file (e.g., the name of an environment). Also, an operation may use another operation defined at the same stage, and the operation used is not persistent (e.g., it had been forgotten). In all these cases changes are performed on the definition file.

Perform Substitution Step (3.3.4)

Instances of user-defined types and operations are substituted by calls to their persistent definitions. When analysing further prototypes, this is only performed in the new or changed modules.

Copy module into a transformed module (3.3.4.1). A module (file) is copied into another module (file). The substitutions are performed in the transformed module while the original remains intact.

Substitute user-defined types (3.3.4.2). If a user-defined type is used in the module, its old name is searched for and, if found, it is substituted for by the new one. The search command in an editor is helpful for this. The module's definition file and the user-defined type's record in the analysis document contain the information of whether the user-defined type was identified in the module or not, and of its old and new names (**OLD NAME** and **NEW NAME**). It should be noted that in some cases the user-defined type is used in the module (i.e., operations on it are identified) although its name does not appear explicitly.

Substitute operations (3.3.4.3). For each operation recorded as identified in the module, the information in its **CHUNK** field(s) is used to identify instances of the operation. They are identified by looking, in these chunks, for code similar to the body of the operation which is described in **BODY** (bearing in mind that it will be applied to different variable names). The identified code is then substituted by calls to the persistent definition of the operation (described in **CALL**).

Record substitutions (3.3.4.4). For each operation, the pattern of the body found in the module, its substitution, and any arbitrary decisions made are recorded in **PATTERNS OF BODY**, **SUBSTITUTION**, and **ARBITRARY DECISIONS** respectively.

Perform Validation Step (3.3.5)

To validate that the definitions and the transformations are correct, correctly used in the program, and that as a result the program's behaviour is preserved, the program (i.e., the transformed modules) is executed. When there are errors, the analysis document is changed to incorporate any changes that modify its information.

Errors. There can be two main sources of errors. Persistent definitions are wrongly used (or not declared) in the program, or they are wrong or missing in the persistent store. These errors can be just mistakes, i.e., wrong (or unperformed) substitutions. They can also be due to interpretation errors during the identification and/or substitution steps. Errors in transformations can be syntactic errors made during the substitutions. They can also be due to the kind of transformation problem describe in section 5.2.7. When the problem is in the program, modifications should be made to the transformed modules (not to the transformed program) since they are the basis for the next stage. When the problem is in the persistent store, the modifications should be performed locally and dynamically without involving the program or previous definition files.

For further prototypes. I have explained in the step 3.3.1.1.1 that the understanding of what changes and what remains the same in the modules is likely to change as the analysis progresses. As a result, parts of the program that are used from previous prototypes may not be usable after a certain stage. These changes may not be detected by inspecting the program but they are detected when the transformed program is validated. Changes that are not noticed produce execution errors. The usual error is that, instead of behaving as it should, the prototype behaves in the same way as the previous prototype. This is because the program uses operations from the previous analysis that should have been redefined in the current analysis. After the errors are resolved, these changes should be noted and taken into account in the initial step of the next stage.

Result at the End of each Level

At the end of the language level, the analysis document should contain a record for each user-defined type (and all its operations) identified at the language level. This record includes information of all the operations identified on the user-defined type. The file(s) containing the original program should be transformed to a form where all type names are specializations of language type names, all names of operations indicate the functionality of the operation and the user-defined type

it is defined for, and they are accompanied by a use statement indicating the type of the persistent operation being called. The persistent store should contain all the user-defined types and operations that are specializations of language types and operations used in the program.

At the end of the structure level, and thus, of the transformation process, the analysis document should contain all the records of the user-defined types (and their operations) identified at the language and structure levels. The file(s) containing the original program should be empty except for a few calls (preferably one) to persistent operations. The persistent store should contain the rest of the program in its new form. That is, it should contain the user-defined types, operations, and their explicitly declared relations identified during the transformation process.

6.3.4 Perform the Domain Level Analysis (4)

At the domain level all the user-defined types identified at the structure level should be interpreted. This involves the interpretation of their definition and of their operations. Interpretations on all the user-defined types are not likely to be performed sequentially. Some will be interpreted as part of interpreting more complex ones.

The list of user-defined types from the structure level (4.1), and the list of operations on each (4.3.1) are extracted from the records in the analysis document. By making these lists explicit in the analysis document (4.2 and 4.3.2), it is easier to see what has been interpreted and what is still to be interpreted as the analysis at the domain level progresses.

For further prototypes. The analysis is only applied to user-defined types and operations that change or that are new in the prototype. It is possible that new or changed parts of the program contain user-defined types and operations identified in previous prototypes. Everything that remains the same should have the same interpretation as before.

Interpret user-defined types (4.4)

The general criterion is to analyse more complex user-defined types before simpler ones. From the transformation process it is clear how user-defined types are related, and which user-defined types are part of the definition of more complex ones. The interpretation of a more complex user-defined type provides a tighter context in which to interpret its components.

Assign initial interpretation (4.4.1). The Knowledge Level description and the knowledge of the problem solving method used provide the information from which to assign initial interpretations. A user-defined type is interpreted as representing an object from the problem domain or the problem solving method (that is believed to be represented in the program). The decision of which user-defined type is interpreted as representing a given object is likely to be influenced by the knowledge of the program's implementation (although it can also be a guess). If the user-defined type is a subcomponent of another, the context provided by the initial interpretation of the more complex user-defined type is useful information for the interpretation.

Check initial interpretation (4.4.2). This is done using the information in the analysis document about the user-defined type, and its definition. Its definition is described in a explicit graph of dependencies between its subcomponents as a result of the transformation process. The user-defined type may also be a subcomponent of other user-defined types. In this case, it is also part of their definitional graphs.

If the user-defined type does not have other user-defined types as subcomponents, the interpretation is checked by describing what relevant aspects of the object are represented in its type definition, and how they are represented. This description is recorded in the analysis document. If the interpretation cannot be confirmed, it is changed until one that can be confirmed is found. All failed interpretations and subsequent changes are recorded in the analysis document with a description of why they fail.

If the user-defined type is a subcomponent of another one, changing its interpretation is likely to involve changing other interpretations further up in the graph that the user-defined type is part of. This is because the initial interpretation is assigned in the context provided by these other interpretations, and, since it is erroneous, the interpretations that provide the context for it may also be erroneous. It can also happen that the new interpretation is inconsistent with the interpretations further up in the graph.

If the user-defined type has subcomponents, the checking has two parts. First, all its subcomponents are interpreted following the graph of dependencies in its definition in a top-down way. This involves an initial interpretation and its checking for each subcomponent. Second, the results from their interpretation are used to see how the definition of the user-defined type represents relevant aspects of the object being represented. That is, what relevant aspects of the object are represented by each subcomponent and how they are represented. The documentation should describe the aspects considered as relevant for each object (and why).

Interpret Operations on User-defined Type (4.4.3)

The list of operations on a user-defined type and information of each operation is described in the analysis document.

Assign initial interpretation (4.4.3.1). This done by relating its name (which indicates its functionality) to the interpretation of the user-defined type it belongs to. The context provided by its relation to more complex operations (which already have initial interpretations) can also be useful in doing this relation.

Check initial interpretation (4.4.3.2). This is done in a top-down way. The body of the operation (described in the analysis document) and the explicit graph of dependencies between operations in the persistent store are used. The graph shows how the operation is related to other operations, i.e., those that use it,

as well as those that the operation uses in its body. If the checking fails, the initial interpretation is wrong, and another one should be attempted. All the information collected from correct and failed interpretations should be recorded in the analysis document.

If the operation is a language level operation, the initial interpretation is correct if the interpretation of the operation corresponds to the functionality of the language level operation. For example, if an operation is interpreted as getting the first person of a list of people, and the language level operation is `head_of_list` applied to a list of people, then the interpretation is correct. If the interpretation does not correspond to the operations's functionality, the interpretation is erroneous and another interpretation needs to be attempted, this time considering the functionality of the operation. Correct and failed interpretations are recorded in the analysis document.

If the operation is an operation defined at the structure level, the checking involves, first, inspecting the body of the operation to identify what other operations are used in it. In the second part of the checking, operations in the body are interpreted. This involves providing an initial interpretation for each operation in the body and then checking it. This part of the checking is thus, a top-down process. In it, initial interpretations are provided for all the operations that are involved in the graph of dependencies of the body of the operation being interpreted. The process stops when language level operations are reached. In the third part of the checking, the interpretation of each operation in the body is used to see how it contributes to the initial interpretation of the operation. For each operation, it should be possible to identify its interpretation as a relevant part of the interpretation of the whole operation. All parts considered relevant for the interpretation of the operation should be identifiable as interpretations on its operations. The documentation should describe the parts that are considered relevant (and why).

6.4 Summary of PAM

In this chapter I have described PAM, the method developed during the investigation of this thesis. Its aim is to help obtaining a clearer description of an incrementally built experimental AI program. The method provides an organised way of re-structuring and interpreting the program during its static analysis. In the transformation process implementational detail is abstracted away by a process of program transformation at two levels—language and structure. Type information and induction from instances found in the program is used to identify user-defined types and operations which are then made persistent and substituted for in the program. Test sets from dynamic analysis are used to test that the behaviour of the transformed program is unchanged, and thus validate the definitions and the transformations. The user-defined types and operations identified during the transformation process are interpreted in terms of the problem domain or the problem solving method used at the domain level. The method incorporates a step to use results from previously analysed prototypes in the analysis of later ones.

Chapter 7

Assessment and Discussion

In this chapter I will discuss and assess the results of the experiments (chapter 5). This involves discussing PAM (chapter 6) and the abstraction constructs used in terms of their supporting role, limitations, and comparison with other possibilities of providing the same support for the analysis of AI programs. I will start by reviewing the aspects that are important for the analysis.

7.1 Important Aspects for the Analysis

Information hiding, increased precision, complexity management, renaming, and documentation are important aspects for the analysis. They are important in assessing the practical applicability of PAM and in understanding the use of abstraction constructs as part of it. Information hiding, increased precision, and complexity management were discussed in section 4.1, and investigated in the experiments (chapter 5). Here, I will further discuss them in the light of the results from the experiments. Two new aspects, identified during the experiments, will also be discussed: renaming and documentation.

7.1.1 Information Hiding

Two kinds of information hiding are important in the transformation process: hiding implementation from definition and use, and separating functional contexts in a program.

As I discussed in section 4.1, information hiding separates the issues of implementation, specification and use. This distinction is important because it allows us to understand better why a program works in the way it does. The experiments have shown that, by providing separate contexts for each of these, information hiding helps in understanding whether something refers to a design decision or an implementation decision.

The experiments also show other ways in which information hiding is important. It is helpful in understanding and correcting misinterpretations. Separating the three issues (definition, implementation and use) helps to understand the origin of a given misconception, and what needs to be changed and why. Aspects of the implementation, specification and use can be processed separately. In the case that the implementation of an operation needs to be changed, the specification and use need not be affected. When it is the use that needs to be changed, the other two elements need not be affected. Finally, if the specification is wrong, it may or may not affect one or both of the other elements. Thus, it increases the precision of the transformation process. As we know from chapter 5, this kind of information hiding has been provided in PAM by persistence (section 7.2.1). Abstract data types (section 7.2.5) and polymorphism (section 7.2.4) were rejected during the experiments.

We saw in section 4.1 that separating functionalities in a program (in modules) is another important form of information hiding. The experiments confirm the discussion in section 4.1, and also show that user-defined types and operations can be interpreted more precisely in the context of a module. Also, the same user-defined type may be interpreted differently in different modules. Some operations might make sense in a given module but not in another. By providing a tighter context than the program as a whole, modules can also help to identify and to correct misinterpretations, thus providing further increased precision.

7.1.2 Increased Precision

In section 4.1 I discussed the importance of increased precision to obtain reliable interpretations during the analysis. An important question that can be asked

after the experiments is: what is the degree of reliability achieved with PAM? How reliable is the identification of user-defined types and operations? Does this transformation process guarantee the correctness of the interpretations placed on a program? If it doesn't guarantee correctness, what degree of reliability does it offer, and what is the basis for that reliability?

PAM does not guarantee correct interpretations in that there is no formal proof behind it. The method is intended to help researchers to re-structure and understand better a program they have developed as part of their research, not to prove some formal property of it. However, the method ensures that the interpretations are reliable and it also ensures that the transformations performed in a program as a result of the interpretations are correct. It does this by incorporating the validation step where increased precision is achieved by the use of type checking facilities (discussed in section 7.2.2) and by checking that the original behaviour of the program is preserved during the analysis.

7.1.3 Renaming

Understanding a program involves a process of renaming. A user-defined type is renamed, at least once, during the analysis. Renaming is important to accommodate new interpretations on user-defined types: new names make explicit the understanding of what a user-defined type represents at each level of the analysis.

For this renaming to take place, it is important to understand the way an object in the persistent store and a subject in a program are bound together. For the binding to take place, the subject must have some kind of description of the object. [Morrison *et al* 87] use types to provide such a description, and they explain that the types of the subject and the object must match by some rule. Structural equivalence is an appropriate type matching rule for this, and it has been used during the experiments to support renaming in PAM.

7.1.4 Documentation

Documentation is important to keep a record of the data structures and operations found in the program—their definition, implementation, and use—and also of the transformations performed. It helps to perform the analysis: it keeps information from previous levels and stages which is used to perform later ones. It also describes the understanding of the program resulting from the analysis. This information is kept in the analysis document.

Information is added to the analysis document incrementally, at each stage and at each step of each stage. Definition files are also a complementary form of documentation. They contain an incremental historical record of the transformation process, i.e., and module(s) were elements defined, as well as all the incremental and dynamic changes performed at each stage.

7.1.5 Complexity Management

We saw in section 4.1 that two kinds of complexity need to be dealt with during the analysis—structural and managerial. The experiments show that, as the complexity of the programs analysed increases, this aspect can became crucial. Already in a medium size program (i.e., the *Mu Torere* program) both kinds of complexity were significant.

We saw in section 4.3.3 that Software Engineering developments are aimed to support complexity management. The abstraction constructs used in PAM to provide information hiding (persistence), increased precision (strong typing), and renaming (structural equivalence) support complexity management: individually (as we will see in sections 7.2.1, 7.2.2, and 7.2.3) and by its coherent integration in a programming language (i.e., not having to worry about integrating their manipulation).

7.2 The Role of the Abstraction Constructs

We have seen in the experiments that not all the abstraction constructs investigated successfully support all the aspects just reviewed. Here, I expand on the results of the experiments for each abstraction construct in terms of the previous aspects.

7.2.1 Persistence

[Morrison *et al* 88] describe persistent programming as a programming paradigm that may be used to control complexity when constructing large systems, and systems with adaptive data. Some important ideas behind the need for persistent programming as a way of controlling complexity are that: when building large systems complexity must be controlled to allow the user to concentrate on the application rather than on the construction; the best known method of controlling complexity is abstraction where the details of a particular level may be ignored when viewing the system from a higher level; and persistence is an abstraction mechanism that helps in controlling complexity by providing the user with a uniform model of data for both short and long term data.

The use of persistence in this thesis is not in the construction of a large software system in the sense of generating a program (or set of programs) for the first time. Instead, I am concerned with the analysis and transformation of an already built program. Nevertheless, in the transformation process a large system is built (or rebuilt) and persistence helps this process by providing information hiding and complexity management.

Persistent programming embodies two of the requirements identified in [Atkinson & Morrison 88] for persistent data:

- The rights to transience or longevity are completely independent of the type of data.

- The types and bindings used should allow system evolution on an incremental and localised basis.

The first requirement implies that any type of data has the right to be persistent. This is crucial for the transformation process, where all kinds of user-defined types, and operations are made persistent.

In relation to the second requirement, [Atkinson & Morrison 88, page 6] state the importance of dynamic bindings for large scale systems, including the cost that static binding has in such systems because changes to program or forms of data require a recompilation of the whole system to re-establish the bindings. For these reasons [Morrison *et al* 87] investigate the nature of binding mechanisms and the need for mechanisms that provide some form of dynamic binding for persistence. This is important for the transformation process because it makes use of dynamically growing data, and dynamic and local modifications to the user-defined types and operations.

Both requirements together allow the evolution of programs incrementally and locally. Changes to incorporate the new type and operation definitions (interpretations) into the program and the persistent store can be performed at each stage of the transformation process, thus supporting complexity management. A kind of persistence that embodies these requirements is a very convenient tool for information hiding, as I will illustrate next.

In the identification step a set of user-defined types and their operations are first defined. For example, at the language level, stage one, `struct1` and some indexing operations, `index1-a`, and `index1-b`, might be identified. Operations are added to the persistent store in the persistent object definition step, at the end of which most operations of this stage will have been identified and made persistent. However, as I will show next, the following steps may still identify some new operations or indicate the need for modifications to the already persistent operations.

In the substitution step the program is transformed using the persistent operations defined. After the program is transformed, it may be found, by

looking at the code in the program, that there are still operations on certain user-defined types that haven't been identified. For example, it may be found that an operation to `create` instances of type `struct1` had been forgotten in the identification step. Allowing incremental addition of objects to the persistent store means that in order to add this newly identified `create` operation to the set of persistent operations, it is not necessary to restore the state of the whole persistent store again when including the new operation. That is, the bindings don't need to be created again, including the new one, as it would be the case if the operations were kept in a static store like modules or libraries. Instead, with this kind of persistence we can define and make persistent this operation on its own, thus keeping the other definitions untouched in the persistent store, and adding the new ones when necessary.

During the validation step some compilation errors may indicate that certain operations need to be defined that hadn't been recognised at the identification step. For example, we could have this instruction: `s1(a) := 3`, and having identified `s1(a)` as an instance of `index1-a`, it could be substituted by `index1-a(s1) := 3`. This transformed instruction would give a compilation error indicating that the result of a function `index1-a(s1)` cannot be on the left hand side of an assignment. From this we can see that the instruction is an `assignment` operation on `struct1`, and not an `indexing` operation as thought at the beginning. Therefore assignment operations need to be identified for `struct1`. As in the previous case, these operations can be defined and added to the persistent store incrementally.

I have illustrated the importance of the dynamic nature of the persistent store for the addition of new bindings during the transformation process on a localised and incremental basis. Next, I illustrate the use of dynamic bindings in the persistent store to modify already existing persistent bindings.

During execution, an execution error can show that the persistent definition of some operations and their use in the program are different. For example, a creation operation on `struct1` might be defined as:

```
create1::  string ; integer → struct1,
```

and in the program it is used from the persistent store as:

```
create1::  integer ; string → struct1.
```

This will produce an execution error indicating that the type system cannot find the operation in the persistent store because the definition of the operation in the persistent store and its use in the program are different (the type of its arguments is different). If the reason for the difference is that the persistent operation was defined erroneously, the definition of the persistent operation has to be corrected. Again, to change the binding in the persistent store it is not necessary to restore the state of the whole store to include the change. Instead, modifications can be made on the already existing bindings on their own, without involving any other bindings. Again, this illustrates how modifications to persistent bindings can be made both locally and incrementally. On the other hand, if the error is in the use of the operation in the program, it is only necessary to change the program, not the persistent store.

The way a binding in the persistent store is modified depends on its mutability (exactly in the same way as a binding in a program). If the binding is constant, then to modify it we will need to remove it and create another binding with the new value. If, on the other hand, the binding is variable, we only need to reassign the new value to the old binding. The requirement for incremental dynamic bindings in the persistent store allows the use of either of the two methods. Personally, I prefer to use constant bindings on operations to avoid unwanted modifications. It is easier to reassign a new value to a variable binding by mistake than it is when the binding has to be removed and created again explicitly in a constant binding.

This shows that the two requirements for persistence make it a very convenient way of providing information hiding on a local and dynamic basis. Next I will illustrate the importance of being able to access the hidden information in a uniform and easy way.

User-defined types and operations defined and made persistent at one stage are used in more than one transformed version of a program and at more than one level. First, they are used to transform the program where they were identified.

For example, the operations on `association_list1` defined at the language level are still used in the transformed programs at later stages of the language level. Second, they are used when defining operations at the structure level as illustrated with the `index1-a` and `second2` operations for `access1-a`. Persistence allows access to the persistent objects by different programs at different times in a uniform and easy way.

Thus persistence helps in performing the transformation process by helping the management of user-defined types and operations found at different stages and at levels of the transformation process. It does this by:

1. Allowing objects and operations to be accessed by different transformed versions of programs.

2. Allowing the definition of operations and objects in the persistent store to be added to and/or modified dynamically from different programs and at different times without regard to when and where they were initially defined.

3. Maintaining the consistency of the use the programs make of the persistent store. The system does this by type checking (partly statically and partly dynamically) the programs against the persistent store.

7.2.2 Strong Typing

Although correct interpretations of user-defined types and operations are not guaranteed by the transformation process, strong typing provides increased precision during the transformation process (together with checking that the behaviour of the program is preserved). It does this by ensuring that all expressions in a program are type consistent. That is, it ensures the reliability of the representation of objects in a program with unambiguous values in the semantic domain provided by the type system. I illustrate the use and importance of strong typing in the method with an example. At one stage of the language level, `(3,z)` might be interpreted as an instance of `list1`, where `list1` is

interpreted as only having two elements: the first of type `integer` and the second of type `string`. Using the facilities of a strong typing system a user-defined type for `list1` can be easily defined. Then `(3,z)` gets transformed into `list1(3,z)` in the program. The indexing operations performed on `(3,z)` get transformed into calls to well-defined indexing operations: well-defined in the sense that the type of the arguments and the results are specified (e.g.,`(car (3,z))` gets transformed into `index1-a((3,z))` where the type of the argument is `list1`, and the result is an `integer`).

Now, say that the interpretation of `(3,z)` was wrong, and that instead of being an instance of `list1`, it is in fact an instance of `list4` (where `list4` accepts `integers` or `strings` as the types of the first element, instead of only `integers` as in `list1`). This means that when the program is executed and a `string` is placed in the first field of this instance, the execution of the program will produce an error because what had been transformed into an instance of `list1` is in reality an instance of `list4`. This error produced by the type system shows the interpretation error of the researcher. Another example could be if `(3,z)` is not really an instance of `list1`, but, say, an instance of `list5`, where `list5` sometimes has more than two elements. If this is the case, during the analysis of the program operations indicating this difference will be found, e.g., an operation that adds a third element to `(3,z)` or one that indexes the third element. When that operation is transformed into a call to the corresponding indexing operation, a strong type system will give an error indicating that something of type `list1` cannot be indexed in the third element (because it can only have two elements). In the extreme case a strong type system can help to understand that `list1` is not really a user-defined type of the program because all the instances taken as examples of `list1` were shown to be instances of other user-defined types. In this way a strong type system can help to develop interpretations of the program being analysed. These interpretations are more reliable than those obtained just from what the researcher thinks is in the program. For this reason strong typing is an important abstraction construct.

Another way in which strong typing can help the analysis is when the program

was already developed using a strongly typed programming language. In this case, some of the problems of interpretation that may be faced during analysis would not occur. If the programming language used to develop the program was a weakly typed language like Lisp, at the language level the interpretation of `list1`, `list2`, `list3`, etc., as defining one kind of `list` or another depends strongly on the understanding of the researcher. This is because in a weakly typed language the definition of the types used in a program is not required. However, in a weakly typed language although the developer is not forced to declare the elements used in the program, he or she can still use typing facilities provided by the language to make his or her intentions clearer. For example, in Lisp I might have a definition of `struct1` as `(defstruct struct1 a b)`, so that interpreting `(make-struct1 :a 3 :b z)` as an instance of `struct1` is again rather straightforward. As a result the program would be easier to analyse. This is not the case with other data structures in Lisp, like `lists` or `association lists` where their definition is not required by the type system. If the language used to develop the program is a typed language where the types need to be explicitly declared, it is more likely that the user-defined types are already explicit in the program. (See section 2.1.1 for a discussion on the use of type inferencing to help in pre-analysing a program written in a weakly typed language.) In Napier88, I am likely to find `struct1` explicitly defined in the program (although it may have a more domain dependent name like board) as:

```
type board is structure(a:integer ; b:string).
```

Thus interpreting `struct(a:3 ; b:z)` as an instance of `struct1` is again relatively straightforward.

To summarise, strong typing supports the definition of user-defined types and operations and the checking of their reliability and the program transformations.

7.2.3 Structural Equivalence

Structural equivalence is a good way of supporting renaming: it provides a way of changing the name of user-defined types in the transforming program without

affecting the definition of operations kept in the persistent store so long as the structural definition of the user-defined types does not change. Consider the example of `association_list1` described in section 6.1.1. As I explained there, at the structure level `association_list1` might be interpreted as representing `struct1`. To reflect this new understanding I only need to rename it as `struct1` in the program at the structure level, but without affecting its type definition.

The persistent indexing operations on `association_list1` at the language level are defined in terms of the name `association_list1`. If the type equivalence of the binding between a subject in a program and an object in the persistent store was based on say the name of the user-defined type (name equivalence) it would be necessary to change the definition of operations like `index1-a` and `index1-b` to incorporate the change in that particular name. A program using `index1-a` at the language level might use the following definition:

```
index1-a::  a11:association_list1 → list2.
```

A program at the structure level though, would use a different definition. The type of the parameter would no longer be `association_list1`, but `struct1`:

```
index1-a::  a11:struct1 → list2.
```

Furthermore, to be able to execute the different versions of the program (for instance, the one that uses `index1-a` with `association_list1`, and the one that uses it with `struct1`) without structural equivalence it would be necessary to maintain copies of all the operations with the different names. This would involve keeping separate persistent stores (or separate environments in the persistent store in the case of Napier88) for all the operations at each stage. The complexity management of the transformation process would increase a great deal if this were the case. Structural equivalence avoids all these problems. It provides the means to change names of user-defined types in an executable program without affecting previously defined operations, so long as the type definition does not change. Thus, it supports renaming and complexity management.

7.2.4 Polymorphism

The use of parametric polymorphism to provide information hiding was investigated in the stack example (see section 5.1.4). However, its use provides generalization, i.e., it supports the notion of abstracting a program to greater degrees of generality. This generality is opposed to the specialization looked for in the analysis.

To understand what particular data structures represent, or, more usually, to confirm that they actually do the representational job that the program builder supposes them to do, implementational details which are not essential need to be abstracted away. The issue of abstracting from implementational detail is thus tied up with the question of representation. We also saw in section 3.3.2 that the question of representation is tied up with a process of specialization (rather than generalization), and that, during the analysis, it is important to identify these specializations. They provide a clearer (more specific) semantic interpretation of the data structures and their operations, thus, helping to understand their representational role in the program. For example, a function which returns the length of an object, could be applied to two specializations of the data structure list. The analysis aims to understand statically to which type of list the function is applied. To understand what each type of list represents it is necessary to know what are the operations performed on each.

A different kind of abstraction is achieved when more degrees of generality are applied. Generalising to more abstract forms doesn't really tackle the essential problem of what the specialized data structures represent. Parametric polymorphism hides the specific data types that can apply to a (polymorphic) structure or operation. That is, it generalizes over types or operations to a uniform structure or behaviour that is applicable to an infinite number of specific data types. In other words, it hides semantic information from the program that is important in understanding the representational role of each specialization. It is not until the program is run that this semantic information can be identified, and thus interpreted. This means that the static analysis won't give as much semantic information as I am interested in achieving if polymorphism is used

during the analysis. Thus, parametric polymorphism does not support this form of analysis as much as it would be desirable. It is for this reason that parametric polymorphism was discarded as one of the abstraction constructs to be used in the transformation process.

As we saw in chapter 4 the other kind of universal polymorphism is inclusion polymorphism. Although it was not investigated because it wasn't available, an interesting distinction can be made between these two kinds of polymorphism. Inclusion polymorphism hides information in a different way. It avoids the repetition of information that pertains to various objects that are semantically related to one class (or more). Instead, the (shared) information is accessible in a class and the objects can access it via the semantic links that inclusion polymorphism provides. The information is thus accessible to the static analysis (it is somewhere in the program), and on top of that, inclusion polymorphism supports identifying (also statically) the semantic dependencies between objects in the program. Thus, it is likely that universal polymorphism could be useful in the static analysis of programs (most of all those built in an object-oriented style).

7.2.5 Abstract Data Types

Data abstraction was investigated because, *a priori*, it can be a good way of encapsulating the information pertaining to the user-defined types and their operations. Data abstraction is conceptually more appropriate to the definition of user-defined types and their related operations than procedural abstraction. It also provides mechanisms to ensure access through appropriate interfaces, thus helping to ensure a correct use of the identified components of a program. However, abstract data types are not used in the transformation process. The incremental nature of the analysis makes it difficult to decide when a user-defined type has been completely understood so that an abstract data type can be defined for it. It is difficult because it is not possible to guarantee that what constitutes the abstract data type (its components, and the definition and implementation of those components) will not change after the abstract data type is defined.

Some user-defined types may get identified in just one stage, e.g., `struct1` and all its operations. In that case an abstract data type could be defined for `struct1` at the end of the stage. However, this is not always the case. Sometimes, more than one stage is required to identify all the operations on a user-defined type. For example, `list1` can have basic list operations that are identified in an early stage, like `head` and `tail`, and operations that are identified in a later stage, like `member` and `print`. Moreover, an operation that was thought to correspond to a certain user-defined type may later be discovered not to be so.

A solution could be to define intermediate abstract data types that keep changing as the transformation process progresses. However, I do not consider this solution a practical option. Performing these kinds of changes would only add management complexity to the already complex task of performing the analysis. Thus, if abstract data types are to be used, it is necessary to define them when the chances of their redefinition are minimal. In the experiments I performed I could not find an optimal point for the definition of abstract data types (except, of course, at the end of the transformation process). The end of each level is also a possibility, although only in some cases. User-defined types that are data structures provided by the language are likely to have operations at different levels (`list1` can have operations at the language level `head` and also at the structure level: `print`).

Abstract Data Types, in principle, could be useful to package each data structure with its operations. However, its use is impractical because of the complexity it adds to the transformation process. Better understanding of their practical application is needed before they can be applied successfully.

7.3 Discussion

7.3.1 Information Hiding

PAM provides good information hiding. It does this by separating definition, implementation, and use, and separating functionalities. Information hiding is

provided on a dynamic and local basis. Hidden objects are accessible by different programs at different times in a uniform and easy way. The kind of information hiding achieved by abstracting over the type of data was not found to be good for the analysis (which involves a process of specialization, not generalization) and thus, it is not attempted in PAM. At the moment, PAM does not support the information hiding achieved by a hierarchical representation with subtyping and inheritance because it was not possible to investigate it. Thus, it remains to be investigated in the future.

7.3.2 Increased Precision

PAM supports good increased precision of definitions and transformations. Their reliability is checked syntactically and semantically so that the behaviour of the transformed form of the program remains the same.

7.3.3 Renaming

PAM supports renaming in a principled way. It ensures that a user-defined type definition is consistently used regardless of the different names given to it. This consistency is maintained in the program as well as in the hidden store.

7.3.4 Documentation

PAM supports documentation in that it identifies what should be documented during the analysis, when, i.e., in which step, stage, etc., and how to use the information in the analysis document to support the analysis (as we have seen in chapter 6).

One type of information that is not recorded at the moment in PAM is variable names of instances of user-defined types identified in the program (during the identification step). Variable names can complement the context information used to facilitate the substitution step (i.e., module and chunk information). It should be recorded as part of the context information, as the same variable name can represent different elements in different contexts.

Information is recorded in the analysis document by hand. Thus, it is up to the researcher to make sure that all the necessary information is kept appropriately. Although it has been sufficient for the experiments, better ways of keeping the information obtained from the analysis need to be investigated. Some of the research in the area of software maintenance and reverse engineering is concerned with documentation recovery (also called re-documentation) of computer programs. In particular they are concerned with the development of tools that help in keeping and visualizing the performed redocumentation. These kinds of tools could be useful for this analysis, and future work on the method should involve investigating their use for documentation.

7.3.5 Complexity Management

PAM supports complexity management in various ways. Structural complexity is supported by the steps, stages, levels, etc. of PAM, i.e., the analysis is performed in a structured way. It is also supported by providing information hiding, increased precision and renaming. Information hiding tightens the context of the analysis, thus, fewer things need to be considered at any one time. Providing it on a dynamic and local basis, and with a persistent, uniform, and easy access to the hidden elements reduces managerial complexity. Increased precision releases the researcher from the verification task, thus, there is less to do. Renaming supports structural complexity without adding managerial complexity.

7.3.6 More on Abstraction Constructs in the Analysis

Next I will discuss some other aspects of the influence of the abstraction constructs in the analysis.

Strong Typing

I have already said that PAM, and in particular strong typing, provides increased precision during the transformation process. Strong typing avoids ambiguity and misinterpretations. However, the semantic domain of a type system is limited to

that of the programming language. It is for this reason that the type checking is only used in the transformation process, not in the domain level analysis.

Strong Typing and AI programs

An explicit strong type system was used for the investigation of this thesis. This means that the programs being analysed already had most types[1] explicitly declared. As I pointed out in section 4.5.1, this makes easier the identification of which types are present in the program. However, even when a strongly typed system is used during the building process, there is a lot of identification to be done during the analysis: how and where user-defined types are used in the program, how they are related and in which contexts, etc. There is the task of identifying which operations operate on which user-defined types, and thus, identifying operations (which may or may not have been declared as procedures[2] in the program).

Explicit type declaration is not required by typical AI languages like Lisp and Prolog (or by some other strongly typed languages like ML) and it could be argued that using an explicit strong type system to build AI programs is not realistic in AI practice. Next I will argue that there is not any real basis for this argument.

Can AI programs be typed? AI languages have type facilities, and they can be used to declare types explicitly. Using them or not is a question of discipline. When they are not used it is up to the researcher to make sure that everything is correctly used (the alternative is to spend as much time debugging the program as building it, if not more). Lisp and Prolog manuals encourage users to make use of the typing facilities as good programming practice (see for example [Ritchie 91b,

---

[1]Basic types like integers, booleans, and others like structures are examples of exceptions in Napier88.

[2]I use the term procedure to mean both procedures and functions.

chapter 6]). Thus, explicitly declaring types is not only possible but also good programming practice.

Can AI programs be built using a strong type system? I showed in section4.4.2 that some strong type systems can be used to build AI programs, i.e., those that incorporate the principle of data type completeness, and a mixture of static and dynamic checking. They provide the expressive power and flexibility usually required in AI programming.

Type information can be extracted from untyped programs. Untyped programs can still be analysed in the same way as typed programs. The type information can be extracted automatically from the untyped program. There is some current research on supporting this kind of extraction. [Frühwirth 88] proposes a simple but powerful meta programming method to derive type information from Prolog programs, and [Palsberg & Schwartzbach 91] proposes a type inference algorithm to infer types in untyped object-oriented programs. The type information can be extracted as a pre-step to the analysis. Type information is thus available for the analysis of initially typed or untyped programs.

Finally, I would like to point out that PAM could still be applied even in the case of not having type information available explicitly (either because types were not explicitly defined or they were not extracted from the program). Given the previous discussion, this point is not likely to be relevant in practice. However, it is an interesting exercise to consider what would happen if PAM was to be applied under these conditions. In this case, the identification step would be more difficult but it would still be possible to perform it. Instead of having the types explicitly defined, they would need to be induced from an inspection of the instances found in the program (as part of the identification step). This is likely to be a difficult task, and one in which erroneous interpretations are more likely to happen because the identification step relies greatly on the analyser. Thus, in principle, PAM could still be applicable when type information is not explicit. However, the analysis of a complex program under these conditions is not (in

practice) a realistic option since we have seen (in the previous paragraphs) that there are easier ways of obtaining the type information.

Structural Equivalence

Structural equivalence might be misleading in the analysis if two entities have similar types, i.e., the same type structure could be used to represent two sorts of entities. This is a limitation in that it would be better (more clear and easier to understand) to be able to distinguish them by some means, i.e., giving a different name to each entity. In this respect structural equivalence does not support the analysis.

The researcher will still be able to distinguish the different entities by looking at the set of operations on the type identified during the transformation process. Different entities are most likely to be operated on in different ways (although some basic operations may be the same, due to their common structure). Thus, it should still be possible to identify each entity and its set of operations. However, it would be much easier if this could be done during the transformation process.

Persistence

The type model of Napier88 allows the persistence of objects and operations but not of data types because they are not values. This prevents the use of persistence in a uniform way during the analysis. The persistence provided by the version of Napier88 used in the experiments lacks facilities to visualize the contents of the persistent store. This makes the analysis more difficult.

7.3.7 Comparison

The question I address here is whether the support provided by the abstraction constructs investigated can be achieved by other means. I compare Napier88 (as a language implementing the abstraction constructs investigated) with more classical AI programming languages such as Lisp [Wilensky 86], and Prolog [Clocksin & Mellish 81]. I also compare the abstraction constructs with other

abstraction constructs like the strong type inference system in ML [Milner 83], the facilities for modularisation in Modula-2 [Wirth 83] and object-oriented languages such as Smalltalk [Goldberg & Robson 83].

AI Languages

Information hiding and renaming facilities can be supported by Lisp and Prolog but they do not provide good facilities for reliability checking or complexity management. I illustrate this conclusion with a small example where I compare the use of Lisp, Prolog and Napier88. In this example I show first how information hiding on a dynamic and local basis can be achieved in the three languages.

In this example, `op1` and `op2` (which makes use of `op1` in its definition) are identified as operations on the user-defined type `list1` at different stages of the transformation process. They are identified as operations that get the first and second element, respectively, of an instance of `list1` (see figures 7–1 and 7–2).

During the analysis, the definition of both operations is removed from the program, and it is put in the definition file (`try1` for `op1` and `try2` for `op2`) together with all the other operations identified at the same stage. Figure 7–1 shows that we can load (in Lisp) consult (in Prolog) or compile and execute (in Napier88) `try1`, so that `op1` can be used in `op2` (see figure 7–2). Thus, information hiding is achievable in the three cases by a similar process.

Let's see how local and dynamic this information hiding is in each case. Say we want to change the definition of `op1` at a later stage so that it now removes the first element of a list, and returns the resulting list (see figure 7–3 for the new definition in each language). If dynamic and local changes were not possible, we would need to go back to `try1` change `op1` and load, consult (or reconsult) or compile and execute `try1` again. Thus, we would need to remember exactly the file in which `op1` was first defined; in modifying `try1`, there is a risk of modifying other operations on it, and changing `op1` results in all other (unchanged) operations being loaded, consulted, or compiled and executed again (except if reconsult is used in prolog instead of consult). However, none of this is necessary in these

LISP	PROLOG	NAPIER88
⋮	⋮	⋮
(defun op1 (list1) (car list1))	op1([X|_T],X).	let op1=proc(l1:list1 → integer) first(l1) in ps let op1 := op1
⋮	⋮	⋮
load try1.lisp	consult try1.prolog	compile and execute try1.napier88

Figure 7–1: Comparison between AI languages and Napier88: program try1.

LISP	PROLOG	NAPIER88
⋮	⋮	⋮
(defun op2 (list1) (op1(cdr list1)))	op2([_X|T],Y) :- op1(T,Y).	let op2=proc(l1:list1 → integer) use ps with op1:proc(list1 → integer) in op1(rest(l1))
⋮	⋮	in ps let op2 = op2
load try2.lisp	consult try2.prolog	compile and execute try2.napier88

Figure 7–2: Comparison between AI languages and Napier88: program try2.

languages, as local and dynamic changes are possible in all of them: it is possible to define a new `op1` in a separate file, `try3` (figure 7–3) whose loading, consulting, or compiling and executing results in the the new `op1` superseding the previous one, without disturbing any other operations.

However, persistence in Napier88 provides a long lasting information hiding not provided by Lisp and Prolog. In figure 7–1 we can see how `op1` is made persistent in Napier88, and figure 7–2 shows the use of the persistent `op1` to

LISP	PROLOG	NAPIER88
(defun op1 (list1) (cdr list1))	op1([_|T],T).	use ps with op1:proc(list1 → integer) in op1 := proc(l1:list1 → list1) rest(l1)
load try3.lisp	consult try3.prolog	compile and execute try3.napier88

Figure 7–3: Comparison between AI languages and Napier88: program try3.

implement `op2`, and how `op2` is made persistent. Changing the persistent `op1` only requires the modification of its binding in the persistent store (figure 7–3) and persistent operations can be used at any other time, and in any other program. This is not the case in Lisp or Prolog. When a session in Lisp or Prolog is finished, all the operations that were loaded or consulted are removed from the working space. The only way to make them persist after a session is by saving the image of the current session (see section 4.4.1 for a discussion of the difficulties of this *adhoc* kind of persistence). If the image is not saved, recovering the definitions of the operations involves loading or consulting all the definition files again, and in the same order (to make use of the dynamic and local changes) for each session. Considering the several stages, and various dynamic and local changes at each stage, having to remember (and record) all the files, the order in which they were created, and loading them again for each session adds a lot of complexity to the practical application of the transformation process, a complexity that is avoided when persistence is used.

Lisp and Prolog are both weakly typed languages, and it is mostly up to the user of the language to use data structures consistently when building more complex structures with them. Reliability checking is better supported by strongly typed systems which provide powerful type constructors with which to build complex structures that are then typed checked by the system.

The same applies to the support provided for the renaming process. Renaming is possible in Lisp and Prolog. However, this is not because the language's type system has a type equivalence rule (such as structural equivalence) which controls the consistent used of renamed user-defined types. In Lisp and Prolog renaming comes for free with the lack of a strong type system, but the system does not control that the different names correspond to the same underlying structure.

Thus, classical AI languages such as Lisp and Prolog do not provide the support given by a language that implements the abstraction constructs investigated in this thesis.

Other Abstraction Constructs

The previous discussion assumes that AI languages do not implement modern abstraction constructs like the ones investigated in this thesis. Some modern versions of these and other programming languages do implement some kinds of abstraction constructs. Here I discuss abstraction constructs that are similar or related to those I have investigated. I discuss them in terms of their supporting role for the aspects mentioned at the beginning of this section.

Type Systems: Most languages that incorporate a type system could support some degree of reliability checking. However, many of those type systems are purely static (e.g., those of Pascal [Wirth 71]), Ada [Ichbiah *et al* 79], Modula-2 [Wirth 83], and ML). They do not allow dynamic binding, they distinguish between program and data, and they do not have structural equivalence to support renaming. Thus, just providing a type system is not enough. A type system has to provide a mixture of static and dynamic checking, first class rights to all types, and a type equivalence rule that supports renaming without introducing further complexity when used.

Modularisation: Modularisation supports information hiding, and languages that provide some kind of modular abstraction, e.g., procedural abstraction, provide information hiding. However, a more strict understanding of information hiding involves actually hiding definitional information away from the program where it is used. This is also called modularisation.

Many modern languages implement facilities for modularisation. In Lisp and Ada they are called packages, and in Modula-2 and Perlog [Moffat & Gray 88], they are called modules. However, packages in Lisp are different from packages in Ada, and the same applies to modules in Modula-2 and Perlog. The kind of type system of the language affects the modularisation facilities provided. For example, packages in Lisp are just an interface between the names used in a program and their internal representation. It allows the division of a program into semi-independent chunks, where the same name can represent different symbols

in different packages [Wilensky 86]. This can result in programs that are difficult to understand and debug because it is up to the researcher to make sure that names are used consistently. Ada and Modula-2 are well-known for incorporating modularisation in a consistent way. In Ada and Modula-2 modules are linked persistently to the program (or programs) that uses the modules from the moment the program is compiled. However, the type system in Modula-2 and Ada is static. Thus, dynamic and local changes to elements in a module are not possible, which we have seen is important for complexity management. Thus, for modularisation facilities in a language to support information hiding without adding complexity, the language needs to incorporate a strong type system that allows dynamic and local changes to the modules.

Object Orientation: Many modern programming languages incorporate facilities to program in an object oriented style—Smalltalk, CLOS (object-oriented Lisp) C++ (object-oriented C) etc. In [Pascoe 86] information hiding, data abstraction, dynamic binding, and inheritance are described as the four elements that a language must have to support object-oriented programming. I have already discussed how the combination of information hiding and dynamic binding supports the transformation process, and that abstract data types doesn't because it adds complexity. Inheritance is used to define specialization relations between objects, and it would need further investigation to discuss its possible use for the analysis of programs that implement some inheritance mechanism. However, strong typing or renaming facilities such as those provided in Napier88 are not found in object-oriented languages. Thus, although they support dynamic and local information hiding, they do not provide good mechanisms for reliability checking and renaming.

7.4 Required Properties of a Language for the Analysis

Abstraction constructs such as strong typing, structural equivalence and persistence support the practical application of the transformation process in the aspects discussed in section 7.1. From the comparison with AI languages such as Lisp and Prolog, and the consideration of other abstraction facilities we have seen that, at present, no language or environment can fully support all these aspects. On the basis of my investigations and the previous discussion, I next describe the required properties of a language (or a coherent environment) to support the analysis.

- Facilities to support two kinds of information hiding: hiding implementation from definition and use, and separating functional contexts in a program. They have to provide it on a dynamical and local basis. Access to hidden objects by different programs at different times has to be provided in a uniform and easy way. Visualization facilities of the hidden objects should be provided.

- Type renaming facilities that are consistent across unanalysed and analysed (hidden) parts of the program. The facility has to ensure that names used at different levels identify the same (implemented) type definition. Otherwise, operations identified in one level won't be recognized as operations on the same type when the name changes.

- Facilities to support the identification and definition of user-defined types and operations, and to check the reliability of the definitions and transformations.

- Facilities to support complexity management. This can be achieved by providing the previous facilities so that they are easy to use, and also providing them in an environment that supports their coherent use.

7.5 The Role of PAM

In this section I discuss the role of the Prototype Analysis Method for the analysis of experimental programs in AI research. The analysis is aimed at obtaining a Symbol Level description, that is, a description of the representational role of the user-defined types and their operations identified in the program.

To support their identification the method incorporates a transformation process which is aimed at re-structuring a complex program to a form that is easier to understand on inspection. The transformation process is thus to be assessed in terms of its support for the effective re-structuring of a program and the practicality of its application. We saw in the experiments (chapter 5) that the transformation process supports the re-structuring of a program, and that, in the resulting form, user-defined types and their operations are more clearly identified and defined as specializations of data structures and their operations. Although a re-structuring of this kind is not easy to perform in a complex program, the transformation process incorporates the use of abstraction constructs that make its application practical. None of the problems encountered during the experiments pointed to problems in the structure of the transformation process, i.e., problems of performing the transformations at different levels of abstraction, or in the stages or steps of the method. Thus, the transformation process provides an effective re-structuring of a program in a practical way. We saw in chapter 6 that the method also incorporates facilities to support the analysis of incrementally built programs, and in section 5.3 I described their use to analyse an incrementally built program.

To provide a Symbol Level description, the identified user-defined types and operations need to be interpreted in terms of the domain problem or problem solving method used. The method incorporates a further level of analysis, the domain level, at which this interpretation is performed. We have seen in chapter 5 how the domain level analysis was performed in each of the experiments, and how, as a result, a Symbol Level description was obtained. Thus, from these initial

experiments, we can see that, by applying PAM, a Symbol Level description is obtained.

7.5.1 PAM and Control Structure in a Program

In section 2.4.2 I said that to understand the control in a program it is usually necessary to have a prior understanding of the data structures and operations that are being controlled. Here I illustrate this point by showing how the Symbol Level description resulting from the application of PAM can be useful in understanding the control structure of the operation `find_path` identified in the maze experiment (section 5.2.5). From the Symbol Level description of this experiment (see figures 5–17, 5–18, 5–19, and 5–20) we know that a path is constituted of cells, and that `find_path` takes as arguments two cells, a path, and the maze, and it returns a path. We also know that it uses operations on the type of path (`find_path`, `add_cell_in_path`), cell (`cell_is_in_path`, `first_cell_in_neighbours`) and neighbours (`find_neighbours`, `empty_neighbours`, `rest_neighbours`).

Understanding the control structure of `find_path` involves answering this question: how is a path found? The use of `find_path` as part of itself indicates that finding a path is a recursive process, and the use of `add_cell_in_path` indicates that a path is found constructively, that is, that finding a path involves building one (and not testing paths already built, for example). Thus, we can say that a path is found by building one (from the initial cell to the final cell given as arguments). Then the next question is: how is a path built? This question can be transformed into: when is a cell added to the path? To answer this question we look at how `add_cell_in_path` is used. Looking at the body of `find_path` (figure 7–4) we can see that `add_cell_in_path` is used in three places, and it is applied to two different instances of a path (`total_path` and `try_path`) which indicates that the process of building a path really involves building two paths.

From inspecting the code in the places where `add_cell_in_path` is used, we can see that building a path involves a 'look-ahead' process for each cell that is not already in the path. `~Cell_is_in_path` indicates that only cells that are not in the path already can be added to it, thus, achieving the condition

```
in maze_objects_env let success := false
in maze_objects_env let total_path := initialize_path()
rec let find_path = proc(c1,c2:cell ; try_path:path ; m:maze -> path)
begin
 use maze_objects_env with success:bool ; total_path:path in
 begin
 if c1 = c2 then
 begin
  total_path := add_cell_in_path(c1,total_path)
  success := true
 end
 else
  if ~cell_is_in_path(c1,try_path) do
  begin
   let n := find_neighbours(c1,m)
   while ~success and ~empty_neighbours(n) do
   begin
    let c3 := first_cell_in_neighbours(n)
    n := rest_neighbours(n)
    total_path := find_path(c3,c2,add_cell_in_path(c1,try_path),m)
   end
   if success do total_path := add_cell_in_path(c1,total_path)
  end
total_path
end
end
```

Figure 7–4: Body of the operation `find_path`.

that a path cannot contain subpaths. In this process paths are tried following a depth-first search process for each neighbour of a cell (`find_neighbours`, `first_cell_in_neighbours rest_neighbours`). This is done by a recursive process that continues while a path is not found (`~success`) and there are still neighbour cells to try a path on `~empty_neighbours`. Each recursive process of finding a path (`find_path`) is done by adding the cell (for which the 'look-ahead' process is done) to the temporary path (`add_cell_in_path`) and attempting a new path with the neighbour cell being investigated. The 'look-ahead' is only successful when one of the neighbours investigated is the final cell. In this case, the final path (`total_path`) is built with the cells whose look-ahead has been successful.

We can see in this example that, to understand the control structure of a program, it is necessary to understand aspects, like recursion in programming, that have not been the concern of this thesis. However, we can also see that, to

understand the use of recursion, it is necessary to understand the data structures and operations involved in the recursive process. These are obtained by applying PAM. Thus, the Symbol Level description resulting from the application of PAM is important in understanding the control structure of a program.

7.5.2 Limitations of PAM

We saw in chapter 5 that the analysis at the domain level is not an easy task, and that the lack of facilities to visualize the analysis document and the persistent store contribute to this difficulty. There is a need for some kind of integrated documentation support similar to that found in CASE (Computer Aided Software Engineering) systems developed to support program building. Tools that could be considered to overcome this limitation are hypertext or hypermedia systems, or tools developed for re-documentation in reverse engineering.

The reliability achieved during the transformation process is provided by type and behaviour checking. I believe that a higher degree of precision could be achieved if the semantics of each user-defined type and its operations could be defined during the transformation process. The transformation process could be completed by adding a further step between the identification and persistence step where the semantics of the identified elements could be defined. The method has only been tested using a specific language, Napier88, and specific problems (modifications) were caused by its use. Finally, the method supports the analysis of programs built by individual researchers, not by teams, for example. In chapter 8, I will point out future research directions to develop and to test the method further, so that these limitations might be overcome.

7.5.3 Comparison

It would be desirable to compare PAM with other methods. However, this comparison is not possible because no comparable methods were found in the literature. As I discussed in the review of section 3.6, research on the understanding and transformation of programs is related to my investigations.

However none of this research (except for rational reconstruction) is aimed at the analysis of experimental AI research programs. Rational reconstruction is concerned with understanding experimental programs to rationally reconstruct them but there are no methods proposed for how to perform this type of analysis.

7.5.4 On the Automation of the Transformations

From the discussion in section 3.6.7 I concluded that automatic tools to perform vertical transformations are not straightforwardly applicable to this work. Also, in sections 3.3.2 and 3.6.7 I explained why this thesis does not attempt to identify general patterns of programming in AI research programs, and therefore, why the use automatic program recognition was not investigated in this thesis.

However, as part of the transformations, user-defined types and operations are first identified in the program, and then substituted. This process of substitution (which also involves recognition of the elements identified during the identification step) is currently done manually. During the maze puzzle problem, I considered the issue of automating the substitution step. However, it was not completely clear what constituted the process of substitution, that is, how the substitution was to be done (either manually or automatically). I considered that it was necessary to understand first what was involved in performing it before any consideration could be given to its automation.

After the investigations, I believe the substitution step could be automated. The automation would involve having a mechanism that would support the definition of patterns dynamically. It would need to be dynamic because the patterns would be defined dynamically as they are identified in the identification step. It would also involve having an automatic search, recognize, and replace mechanism that would automatically recognize and substitute the patterns in the program. Techniques from automatic program recognition could be used in this part of the method in the future. These techniques could help in identifying and substituting correctly different patterns of the same operation, or an operation applied to different instances (and thus with different names) of the same user-defined type. For the recognition to be really successful, the researcher that

developed the program should be involved in the recognition. In this respect, rather than trying to fully automate the transformations, it would be better, and more realizable, to attempt to develop systems which actively support a researcher during the transformations. The development of *support systems* rather than *fully automated systems* is the concern of research in areas such as design, [Smithers *et al* 89], planning, and diagnosis, and should be taken into consideration in the future.

Other aspects that can also be automated in the method are the comparison of versions during incremental analysis, and the management of definition files during the persistence step, for example. Tools for version control such as SCCS, RCS, and MAKE which support the revision of source programs should be considered in the future to automate these aspects.

7.5.5 Applicability of PAM in other Areas

PAM has been developed for the analysis of experimental AI programs built to create some kind of intelligent behaviour. We saw in section 3.2 that these programs are typically built without clear specifications and by prototyping. We have seen in section 3.2.3 that prototyping is widely used both in AI and Software Engineering. PAM is likely to be applicable in these areas. In particular, the analysis of programs built by incremental prototyping in these areas can benefit from the use of a method that takes into account the building process, not only the final program. Thus, PAM is likely to be useful for the analysis of programs, built without clear specifications and by prototyping, in other areas of AI research, and also in application areas of AI and Software Engineering. For example, an interesting area of application for PAM in Software Engineering is reverse engineering (section 3.6.5). The analysis in this case is likely to be performed by somebody other than the original developer, thus, it will add a further interesting constraint to the investigation of PAM.

7.6 Summary

In this chapter, I have discussed and assessed the results of my investigations. This has involved reviewing the aspects that are important for the analysis, and assessing the support provided by the abstraction constructs investigated to achieve them. I have assessed the support provided by PAM to obtain a Symbol Level description of the program being analysed. I have discussed the limitations of the abstraction constructs and PAM, and I have proposed ways to overcome them. When possible, I have compared them with other means of providing the same support for the analysis of experimental AI research programs. I have also discussed the wider applicability of PAM, and the automation of the transformations performed during its application.

Chapter 8

Back to the Framework

In section 2.5.4, I proposed a framework for performing experiments based on three levels of understanding: Knowledge Level, Symbol Level, and System Engineering Level. In this chapter, I will discuss how the Symbol Level description obtained for each experiment relates to the other two levels of the framework. Relating the three descriptions will give rise to the identification of some interesting situations such as inconsistencies between the descriptions.

8.1 The Stack Experiment

I will relate the Symbol Level description of the stack to its description at the Knowledge Level and the program at the System Engineering Level. From this, different situations are identified that help in understanding the relationships between the three levels. This discussion will provide the basis for further discussion of the relationships between the three levels in the other two experiments.

8.1.1 Knowledge Level Description

The behaviour of a stack is rather simple, and it could be argued that it is not the kind of behaviour that requires a Knowledge Level description. Knowledge Level descriptions are usually descriptions of intelligent behaviour, which is also usually understood to be a more complex behaviour than that of a stack. However,

here I will provide a Knowledge Level description of a stack, because it is useful for the purpose of the discussion in section 8.1.2. The Knowledge Level description I provide for the three experiments is based on Newell's description of the Knowledge Level [Newell 81, page 13]:

> The system at the knowledge level is the *agent.* The components at the knowledge level are *goals*, *actions*, and *bodies.* Thus, an agent is composed of a set of actions, a set of goals, and a body. ... Thus, the agent processes its knowledge to determine the actions to take. Finally, the behaviour law is the *principle of rationality:* Actions are selected to attain the agent's goals.

Figure 8–1 shows the Knowledge Level description of a stack, that is, the body of knowledge, the set of goals, and the set of actions of a stack agent. In this description we can see that by processing its knowledge a stack can determine what actions to take (e.g., by processing the first piece of knowledge in the body of knowledge in figure 8–1, a stack can determine that the action it has to take is **ADD**). In this figure we can also see that applying the principle of rationality is very simple, because the execution of each action results directly in a goal being attained.

8.1.2 Relating Knowledge Level and Symbol Level Descriptions

Given the Knowledge Level description of a stack in figure 8–1, and the Symbol Level description in figure 5–12, page 104, we can observe some inconsistencies between the two descriptions. For example, the body of knowledge indicates that a stack can take data of only one type, whereas at the Symbol Level, two types of data are identified: an abstract type `t` and a concrete type `int`. Thus, we can see that a Symbol Level description can be inconsistent with the Knowledge Level description. This inconsistency is due to the fact that the Symbol Level description includes the description of a particular instance of a stack that is not described at the Knowledge Level. We can also see that an agent stack only has two actions, whereas there are five operations at the Symbol Level. One of the reasons is that the action **RETURN** (return one item) requires a combination

BODY OF KNOWLEDGE

A stack agent knows:

- that it can take data items of one (and only one) particular kind, that its ADD action can be used to do this, and that this results in its collection of items being increased by one;
- that the order in which it is given items must be remembered;
- that it can give back items, but only in the reverse order from which they are given to it, that its RETURN action can be used to do this, and that this results in its collection of items being reduced by one.

SET OF GOALS

A stack agent can have as goals:

- to take in one item;
- to give back one item.

ACTIONS

A stack agent has as actions:

- the acquisition of one item: ADD
- the return of one item: RETURN

Figure 8–1: Knowledge Level description of a stack.

of the operations `top` (give back an item without modifying the stack) and `pop` (remove an item from the stack). Other operations like `createStack` cannot be straightforwardly related to any of the knowledge, goals or actions at the Knowledge Level. However, we would expect to be able to identify them in the Symbol Level because they describe coherently how the particular system created in the program realises the behaviour. These examples indicate that an initial knowledge Level description (built at the start of an experiment) may need to be refined at this stage to be more specific to the particular agent implemented and coherently described at the Symbol Level. This is consistent with Newell's view [Newell 81, page 43]: "Knowledge is a radical approximation, failing on many occasions to be an adequate model of an agent. It must be coupled with some symbol level representation to make a viable view".

We should be able to identify in the Symbol Level description everything found in the Knowledge Level description, i.e., the body of knowledge, goals, and the actions. However, we can see that none of the stack goals are explicitly represented in the derived Symbol Level description, perhaps because the simple

behaviour of a stack does not require the explicit representation of goals which is usually expected following from their explicit identification at the Knowledge Level. In more complex AI programs, it may be that goals are implicitly represented or they might not be present at all (which would imply that the Symbol Level description is incomplete with respect to the Knowledge Level description).

8.1.3 Relating Symbol Level and System Engineering Level Descriptions

Relating the Symbol Level description with the System Engineering Level (the program) is a different kind of process from the one we have just seen because the Symbol Level is already related to the System Engineering Level (it is derived from it). This process involves checking the Symbol Level description, and the differences encountered in its relation with the Knowledge Level, with the program. This is necessary because it could be that the inconsistencies and/or incompletenesses identified at the Symbol Level (with respect to the Knowledge Level description) might not be present in the program. It could be that they are the result of inappropriate interpretations during the analysis process. In the case of the stack, the inconsistencies (two descriptions of the type of data: `t` and `int`) and incompleteness (lack of explicit goals) can also be seen in the program.

Relating the Symbol Level description back to the program is also important for checking how well symbols are implemented in the program. For example, it is important to know how accurate is the implementation of the operations `top` and `pop` for them to be good realizations (at the Symbol Level) of the action **RETURN** (at the Knowledge Level). In the body of knowledge it says that a stack agent knows that it can give back items. This implies that when it is empty it just doesn't give back anything. This situation is not described at the Symbol Level (there isn't an operation that tests whether the stack is empty) because it is not explicitly implemented in the program. This means that although the description at the Symbol Level is that of a stack, it may have an implementation

that is incomplete or inconsistent. Being recognizable as a stack does not mean that its implementation is completely adequate.

8.2 The Maze Experiment

In this section I will discuss the Symbol Level description derived from the maze program with respect to the other levels of the framework. As in the case of the stack, I will relate the Symbol Level description derived from the maze program in section 5.2.5 to the Knowledge Level description and to the program at the System Engineering Level. I will see if the situations identified in the stack (i.e., incompleteness or inconsistencies between the descriptions) are also identified in the maze experiment.

8.2.1 Knowledge Level Description

Figure 8–2 shows the Knowledge Level description of this maze puzzle solving competence: the body of knowledge, goals, and actions. By processing its knowledge (e.g., the first piece of knowledge in figure 8–2) a puzzle solving agent can determine what actions to take (LOOK). Applying the principle of rationality is more complex in the maze than in the stack. For instance, the execution of the action LOOK may not always achieve the goal of finding a path (because a path may not exist). However, it is still rather straightforward to match actions with goals.

8.2.2 Relating Knowledge Level and Symbol Level Descriptions

We can observe that the Symbol Level description (figures 5–17, page 121, 5–18, page 122, 5–19, page 122, and 5–20, page 122) is consistent with the Knowledge Level description (figure 8–2). For example, we have seen that `cell` and its operations represent coherently the points in a maze. If there were two (or more)

BODY OF KNOWLEDGE

An agent solving the maze puzzle knows:

- how to find a path between two points in a maze after the two points and the maze are given to it, and that its LOOK action can be used to do this.
- that a path may not exist in which case it has to say so, and that its action NOPATH can be used for this.
- that when it finds a path, it has to display it, and that its action DISPLAY can be used for this.

SET OF GOALS

An agent solving a maze puzzle can have as goals:

- to find a path between two points of a maze.
- to display a path.

ACTIONS

An agent solving the maze puzzle has as actions:

- the search for a path between two points of a maze: LOOK
- the return of a path: DISPLAY
- saying that a path does not exist: NOPATH

Figure 8–2: Knowledge Level description of the maze.

user-defined types realising points in a maze for example, we would need to check that their (compound) representation was coherent.

We can also see that it is not straightforward to relate some user-defined types (e.g., `neighbours`) and operations (e.g., create a non-empty maze, add a cell at the end of a path, get the first cell in a path, or any of the operations on `neighbours`) to any of the knowledge, goals or actions at the Knowledge Level. For instance, the operation on `cell` to `say if a cell is in a path` which tests for membership of a particular `cell` in a `path` (figure 5–18) is not part of finding a path, unless there is some restriction on the kind of path to be looked for. In the Knowledge Level description there are no restrictions on the kind of path to be found. Thus, what is this operation doing at the Symbol Level? From the analysis of the program, I know that the objective of this operation is to prevent having repeated subpaths within a path, and checking that a cell is not already in a path is a way of doing this. This indicates that knowledge of the restrictions on the kind of path to be found is missing from the Knowledge Level description, and thus, the initial knowledge Level description needs to be refined at this stage

to be more specific to the particular maze solving agent described at the Symbol Level.

Also we can see that the Symbol Level description is not complete with respect to the Knowledge Level description. For example, the action **NOPATH** is not identified at the Symbol Level: none of the operations on `path` have to do with saying that a path does not exist.

8.2.3 Relating Symbol Level and System Engineering Level Descriptions

We have just seen that the Symbol Level description is incomplete with respect to the Knowledge Level description. It is important to identify whether this incompleteness comes from an incomplete program, or if it is the result of an incomplete analysis of the program.

For example, the action **NOPATH** is not identified at the Symbol Level. However, the main program contains code to perform this action (section D.1). The problem is that it had not been identified as an operation on `path` during the analysis. This means that, in this case, the incompleteness of the Symbol Level description is due to an incomplete analysis of the program, not to an incomplete program.

As in the case of the stack, we can relate Symbol Level and System Engineering Level descriptions to see how well symbols are implemented in the program. Having established that there is a symbol that represents the maze mentioned in the body of knowledge, we can ask how well it represents it. We know that mazes are finite structures of arbitrary size. However, in this case the size is known by the agent because the maze is given to it, thus it is not arbitrary. The symbol representing the maze is represented by a `list`, which is a data structure of arbitrary size and it is operated on as such in the program. Thus, the `list` does not successfully represent the definite size of the maze which is known by the agent. This incompleteness of the Symbol Level description is due to an

incomplete program, and not to an incomplete analysis of the program as in the previous case.

8.3 The Mu Torere Experiment

In this section I will relate the three descriptions obtained for the *Mu Torere* experiment. Although, there are two derived Symbol Level descriptions of the *Mu Torere* program, one for each prototype, I will consider the two descriptions as one, except when it is important to distinguish them for the purpose of the discussion.

8.3.1 Knowledge Level Description

Figure 8–3 shows the body of knowledge, the set of goals, and the set of actions of an agent playing the *Mu Torere* game. By processing its knowledge, a *Mu Torere* playing agent can determine what action to take. Selecting the action is easy because there is only one: MOVE. However, applying the principle of rationality is not straightforward. The principle says that actions are selected to attain the agent's goals. The goal is to win and the action to move a *perepere*. The game is not won by just moving a *perepere* (otherwise the first one in moving one *perepere* at the beginning of the game would win). An agent playing the game needs to know how to win, thus it needs to perform some more interesting processing of its knowledge to determine the move that will lead it to win the game. To clarify this point it may be worth remembering that the rules of the game distinguish three kinds of move: move to the *putahi*, move from the *putahi*, and move to an adjacent *perepere*, and in many configurations there is a choice as to what move to make.

BODY OF KNOWLEDGE

An agent playing the Mu Torere game knows:

- the rules of the game, which *pereperes* it is playing with, what the current state of the game is, and whose turn to move it is.
- how to make moves in a given board according to the rules, and that its action MOVE can be used for this,
- how to win (not just how to play according to the rules).

SET OF GOALS

An agent playing the Mu Torere game can have as goals:

- to win the game.

ACTIONS

An agent playing the Mu Torere game has as actions:

- the moving of a *perepere*: MOVE

Figure 8–3: Knowledge Level description of the Mu Torere.

8.3.2 Relating Knowledge Level and Symbol Level Descriptions

In the Knowledge Level description we can see that an agent playing *Mu Torere* knows the rules of the game. See section 5.3.2 for a reminder of these rules, and figures 5–37 (page 158) and 5–38 (page 159) for an illustration of the derived Symbol Level description of the rule base and the operations involved in its creation.

Some elements of the Knowledge Level are easily identifiable at the Symbol Level, e.g., the structures representing which *pereperes* the agent plays with, the current state of the game and whose turn to move it is, are identified explicitly in the Symbol Level description as part of the fact base, some rules of the rule base are directly identifiable as implementing the game rules about the three kinds of possible moves, and the MOVE action is identified in their action part. However, careful observation of the Symbol Level description is required to identify other elements of the Knowledge Level description, like the rules of the game. For instance, the first rule, and part of the third (black always begins) are implicit in the action of the (rule base) rules for the beginning of the game. This means that it is not always easy or straightforward to relate Knowledge Level and Symbol

Level descriptions, as some of the knowledge is implicitly in the symbolic structures and their manipulation.

Both prototypes display the specified behaviour at the Knowledge Level, i.e., they both only make legal moves of the game—if they didn't they would not satisfy the external behaviour specification aspect of the Knowledge Level description. However, in the Symbol Level description derived from the first prototype, it is not possible to identify anything representing the top-level goal of winning the game, either explicitly or implicitly. We can now see that, in trying to interpret the Symbol Level description in terms of the initial Knowledge Level description, the first prototype program cannot be understood as having as a top-level goal the winning of the game. Rather, it is to be understood as having the top-level goal of only making legal moves. This is not surprising given that the first prototype was built only to play according to the rules. It is nevertheless important to establish that this is what it does. In the Symbol Level description of the second prototype, the top-level goal of winning the game is identified as being implicitly in the strategic rules. However, we can see, from the behaviour of the program and the causal structure of the strategic rules, that a more complete understanding of the program leads to an introduction of three subgoals. These are (in order of their priority): to win in the next move if possible, avoid losing in the next move if possible, and otherwise make a legal move. From attempting to interpret the Symbol Level description of the second prototype in terms of the initial Knowledge Level description, we have identified a need to refine the Knowledge Level description to include a reference to the three subgoals, their ordering, and their associated actions. The set of actions would now be: WINNING MOVE, AVOID LOSING MOVE, and MOVE.

8.3.3 Relating Symbol Level and System Engineering Level Descriptions

Incompletenesses at the Symbol Level can be identified as implementational incompletenesses. For instance, the goal to win the game (not identified in the Symbol Level description) is not identified in the program; also, the representation of the board is incomplete in that it is implemented as a list and there is no control in its representation (or the way it is operated on) to limit the number of *pereperes* in it to eight. As it is implemented, the program doesn't care whether the board has eight or a hundred *pereperes*. The colour of the *pereperes* is implemented as a string, and there is no control in the program to make sure that the user only inputs one of three instances of string for the colour: white, black, or empty. Thus, the program controls the correctness of its own moves, but not those

of the user. In that sense, its representation of what constitutes legal moves is incomplete.

8.4 Summary

Even in a simple experiment like the stack, it is easier to identify possible inconsistencies and/or incompletenesses in the descriptions at the three levels when these descriptions are related. Identifying these situations is an important part of experiments. It helps to understand better the experiment performed, that is, its Knowledge Level description, the causal structure required to achieve it, and the program that implements this causal structure. This identification is more difficult as the complexity of the behaviour investigated increases (i.e., it was more difficult for the *Mu Torere* than for the stack). However, providing descriptions at the three levels, in particular Symbol Level descriptions derived from programs, has been shown to be important in relating the descriptions of the three levels, and in better understanding how the implemented program produces its behaviour.

Chapter 9

Conclusions and Further Work

This chapter presents the outcome of investigating the thesis that Software Engineering abstraction constructs can help to understand better AI research programs. We have seen that they help the analysis as part of an analysis method I have developed during my investigations—the Prototype Analysis Method. This outcome is discussed in terms of its implications for AI research in general and the development of programming languages and environments to perform experiments in particular. The implications of this outcome for program understanding outside AI research is also discussed. Finally, some ideas for future work and research directions are outlined.

9.1 Outcome of Thesis

I have investigated how abstraction constructs that are developed in the area of programming language research in Software Engineering can be used to help in better understanding AI research programs. This involved establishing what it means to understand better AI research programs, how that understanding can be obtained in a practical and structured way, and whether and how abstraction constructs can help in this process.

Experiments in AI research need to be understood at three levels—Knowledge Level, Symbol Level, and System Engineering Level. In this framework, programs

need to be analysed to get a better understanding of them, an understanding from which a clear Symbol Level description can be constructed.

I have developed a method, PAM, with which to analyse incrementally built AI research programs. The experiments performed to develop and test this method have demonstrated, first of all, that the analysis of programs of the kind developed for AI research is doable. However, given the complexity that can arise in the process of analysing a complex AI research program, the fact that the analysis can be done is not enough. It also has to be practical, that is, complexity needs to be controlled while the analysis is performed. I have shown how using the PAM, the analysis can be performed in a structured and practical way. This practicality is mostly achieved with the use of Software Engineering abstraction constructs.

The experiments have shown that persistence, strong typing, and structural equivalence help in the analysis and understanding of AI research programs. They play a very important role in making it possible to perform the transformation process in a structured and practical way. I have shown how the transformation process helps in re-structuring a program so that it is easier to understand in terms of data structures and operations, and how the domain level analysis helps a researcher in the interpretation of the program's new form in terms of the problem domain or problem solving method used. The interpreted data structures and operations constitute the new understanding of the program, and thus, of the Symbol Level description.

AI research programs are often built by incremental prototyping. Thus, I have investigated the analysis of incrementally built programs. I have shown how the transformation process and the domain level analysis are modified neatly to make use of much of the analysis performed on previous prototypes without adding further complexity to the analysis, and how the use of abstraction constructs plays a vital role in this easy and neat modification of the transformation process and in its practical use.

In summary, the outcome of the thesis is that abstraction constructs embedded in a programming language used for the development and analysis of AI research programs are very helpful during the analysis.

9.2 Discussion of the Outcome

The outcome of the thesis has certain implications for AI programming languages and environments. I advocate the use of abstraction constructs that need to be embedded in a programming language so that they help in the transformation process of a program. I am also advocating an environment where documentation is kept in a convenient way.

Current AI environments do not have facilities to support the application of PAM. We have seen (section 7.3.7) that the main programming languages currently used in AI research—Lisp and Prolog—do not contain the abstraction constructs I advocate, or equivalent ones. The documentation tools that are available for programming are developed to help the development of programs mostly in Software Engineering—top-down analysis, design, and programming tasks. The exception are tools developed in the area of reverse engineering for re-documentation of programs (as discussed in section 3.6.5). These re-documentation tools however, are not integrated in an environment that supports the analysis of AI research programs.

In general, current languages and environments are not developed as part of a methodology to perform experiments in AI research. As a result, they do not contain facilities to support all the steps of the methodology presented in this thesis. Efforts are made to develop environments to support the Knowledge Level step (e.g., Knowledge acquisition tools) and System Engineering Level step (e.g., programming languages and development tools). At the moment this is not the case with the Symbol Level step. The analysis and understanding of programs at the Symbol Level is an area much overlooked which requires much more attention if AI research is to be effective as a science. Regrettably, current research does not seem to be pointing in this direction. Most of it continues to focus on the Knowledge Level and System Engineering Level.

However, this does not mean that an environment to support the application of PAM cannot be developed. From my investigations, I conclude that the analysis

and understanding of programs is an important research direction from which AI research can gain a lot in terms of methodology and scientific results. For it to be practical, I propose PAM, and for this method to be usable, programming languages and environments need to be developed that contain the abstraction constructs and documentation facilities I advocate.

The development of such an environment can be done by developing programming languages (based on Prolog or Lisp for example) that contain constructs like persistence, strong typing, and structural equivalence as they are available in Napier88, and that integrates documentation facilities.

PAM has been developed in the context of AI research, in particular in the context of experiments performed to create some particular kind of intelligent behaviour. However, as I pointed out in the introduction (section 1) results from this investigation are likely to be useful in the broader context of understanding computer programs in general. In particular, I discussed in section 7.5.5 the applicability of PAM in other areas of AI and Software Engineering.

9.3 Future Work

Many questions still remain unanswered with respect to the understanding of AI research programs and the use of Software Engineering research to help the process of understanding. Here I summarize the questions that have arisen and remain following my investigations.

9.3.1 The Analysis Method

- The complexity management involved in applying the method to a large AI program could be helped by automating parts of the transformation process (such as the substitutions step). It will be of future research interest to see if techniques such as automatic program recognition or the development of a support system (instead of a fully automated system) can be used.

- A better understanding of the practical use of abstract data types in the method is needed before they can be applied successfully to provide a more complete identification of the relation between user-defined types and operations.

- Further development of persistent languages should fully realise the principle of data type completeness so that data types can be made persistent. This is important for a uniform application of the transformation process.

- Better ways of keeping and retrieving the information obtained from the analysis need to be found, and for this research in software maintenance and documentation recovery can be useful.

- A better understanding of the semantics of the user-defined types and operations identified would be useful to provide a higher degree of reliability of the interpretations place on them.

- The influence of a strong type inference system like that in the language ML as well as the use of object-oriented characteristics like subtyping can be of future research interest.

9.3.2 Application of the Method

- Further testing of PAM is required. This involves researchers using the method to analyse their AI experimental systems. At the moment, this is only possible if the program is implemented in Napier88. This is a problem because it is unlikely that researchers would be prepared to develop their system in Napier88 (instead of one of the more usual languages like Lisp or Prolog).

- For the method to be further tested, new environments that support its application need to be developed, environments where programming languages generally used in AI incorporate the abstraction constructs

successfully investigated in this thesis. Testing the method using these new environments could also help in understanding better the influence of the language in the analysis.

- The applicability of the method in a broader context than that of this thesis is an important part of the further testing of PAM. Each context is likely to raise different and new issues. For example, in areas of AI that involve program understanding, like validation of knowledge-based systems, issues such as consistency and completeness of knowledge bases will probably need to be addressed.

9.3.3 Understanding AI Research Programs

- In chapter 8 we have seen that it is not enough to obtain descriptions of an experiment at the Knowledge Level, Symbol Level, and System Engineering Level. Performing experiments also involves relating the three descriptions, and the analysis of the situations arising from it. As part of the framework for experiments, further work needs to be done to understand better the relation between these descriptions.

- Another important area for further work is the investigation of the analysis of programs developed not just by one person, but by a team. Many research efforts involve a team designing and building an experimental system. The analysis of such a system would require each researcher to analyse his or her part. The documentation and the persistent store should be shared by all which is likely to precipitate problems of concurrency and distribution that would need to be carefully tackled. It could also be helpful to solve some traditional problems of team developed programs like maintenance of consistency.

Finally, an important outcome of this thesis is that research in AI could benefit significantly from a greater use of concepts and techniques developed in Software Engineering. We can therefore see that practising AI as a scientific activity will gain if more notice is taken of work in Software Engineering.

Appendix A

Example of Transformation Process

This is an example of the kinds of transformations performed on a program as a result of the identification of user-defined types and operations at various stages of the language level and the structure level. The example corresponds to a printing operation, and it has been taken from one of the experiments performed during the investigation of this thesis, the Mu Torere experiment described in section 5.3.

This example presents the original operation in figure A–1, and the resulting forms of the operations generated during the incremental transformations performed on it (by hand) using the transformation process that is part of PAM (the Prototype Analysis Method developed in this thesis). In this example, two stages are performed for each level. See sections A.2.1 and A.2.2 for the language level, and sections A.3.1 and A.3.2 for the structure level. For each stage, the user-defined types and operations identified are presented, together with the transformations performed in the printing operation as a consequence of their substitution. Figures A–2 and A–3 present the resulting form of the program after stage one and two respectively of the language level, figure A–4 the form after stage one of the structure level, figure A–5 an intermediate form at stage two, and figure A–6 presents the result of the transformation process. Comments in the code are preceded by an exclamation mark (!).

A.1 Original Print Operation

This printing operation, called `print_board`, see figure A–1, prints all the elements of a given list. The type of the list is called `boards` and it is implemented as a recursive combination of the language types `structure`, and `variant` (see appendix C for more on Napier88 syntax):

```
rec type boards is variant(cons:node_boards ; empty:null)
& node_boards is structure(position:int ; color:string ; next:boards) ,
```

where each element of the list (each structure) has two named fields (the fields in a structure are indexed by name) the first is called `position` and it is of type `integer`, and the second is called `color` and it is of type `string`. The pointer to the next element in the list is implemented as a third field of the structure called `next` which recursively points to the `variant`.

```
rec let print_board = proc(board:boards)
 if board isnt empty do
 begin
  writeString("BOARD ")
  writeInt(board'cons(position))
  writeString(board'cons(color))
  writeString("'n")              !return character
  print_board(board'cons(next))
  end
```

Figure A–1: Original print operation.

The operation is implemented as a recursive operation that takes the list (`board`) as a parameter. Each element of the list is printed until the list is empty. Printing each element involves printing a heading which is the string 'BOARD', followed by the fields `position` and `color` of the first element of the list (which is accessed by `board'cons`). Then the operation is recursively called with the value of the pointer field `next`, that is, with the rest of the list.

As this operation is defined as a procedure, its implementation is separated from its use. When it is used somewhere in the program, the form of the call is: `print_board(X)`, where `X` is the actual parameter.

A.2 Language Level

At the language level, all the user-defined structures and variants present in the example are identified, together with operations performed on them (the operations identified are specializations of language operations, i.e., indexing and creation). Structures are identified in the first stage, and variants in the second.

A.2.1 Stage One

In the first stage one user-defined structure is identified in this example (`struct1`) together with three indexing operations on it (one for each field of the structure). The original program is transformed as a consequence of substituting the identified structure and operations on it.

Definitions at Stage One

The user-defined structure identified is defined as:

```
struct1 is structure(position:int ; color:string ; next:boards).
```

Identified operations on `struct1` in `print_board` are three indexing operations, one for each field of `struct1`:

```
let index1-a = proc(s:struct1 -> int) ; s(position)
let index1-b = proc(s:struct1 -> string) ; s(color)
let index1-c = proc(s:struct1 -> boards) ; s(next) .
```

Result of Stage One

The transformed `print_board` after the substitution of the indexing operations can be seen in figure A–2.

```
rec let print_board = proc(board:boards)
 if board isnt empty do
 begin
  writeString("BOARD ")
  writeInt(index1-a(board'cons))
  writeString(index1-b(board'cons)) ; writeString("'n")
  print_board(index1-c(board'cons))
 end
```

Figure A–2: Print operation after stage one of the language level.

A.2.2 Stage Two

In the second stage one user-defined variant is identified (`variant1`) together with a testing operation, and an indexing operation.

Definitions at Stage Two

The user-defined variant identified is defined as:

```
type variant1 is variant(cons:struct1 ; empty:null) .
```

Identified operations on `variant1` are an operation to test if the variant is empty, `vis1b`, and an operation to index the other possible field of `variant1`, `vind1a`, which gets an element of type `struct1` (identified in the previous stage).

```
let vis1b = proc(v:variant1 -> bool) ; v is empty
let vind1a = proc(v:variant1 -> struct1) ; v'cons
```

Result of Stage Two and Language Level

When the type `boards` is substituted by the newly identified `variant1`, together with the operations on it `vis1b` and `vind1a`, the operation `print-board` is transformed into the new form that we can see in figure A–3, which is also the final form at the end of the language level.

```
rec let print_board = proc(board:variant1)
if ~vis1b(board) do
begin
 writeString("BOARD ")
 writeInt(index1-a(vind1a(board)))
 writeString(index1-b(vind1a(board))) ; writeString("'n")
 print_board(index1-c(vind1a(board)))
end
```

Figure A–3: Print operation after stage two of the language level.

A.3 Structure Level

At the structure level I identify the data structure(s) implemented by the previously identified `variant1` and `struct1`. This involves looking at them individually and in combination. In this case what is interesting is the identification of a user-defined type `list1` (a specialization of the data structure list) as a result of the recursive combination of `variant1`, and `struct1`. Thus, in the first stage I identify user-defined `list1`, and the most basic operations on a list performed on it. More complex operations are identified in the second stage.

A.3.1 Stage One

In the first stage the user-defined type `list1` is identified and defined, together with basic operations on a list that are performed on it; operations such as obtaining the first element of the list (`head1`) obtaining the list without its first element (`tail1`) and testing whether the list is empty (`is_empty1`).

Definitions at Stage One

`List1` is defined as a recursive combination of `variant1` and `struct1`. This involves renaming `variant1` as `list1` when the combination is made. Also, the field `next` of `struct1` doesn't point to `boards` as it did before. Now it points to `list1`:

```
rec type list1 is variant(cons:struct1 ; empty:null)

 & struct1 is structure(position:int ; color:string ; next:list1) .
```

Three operations are identified in the printing operation for `list1`. Testing if the list is empty (`is_empty1`) is defined in terms of the testing operation on `variant1` identified in the second stage of the language level. Obtaining the first element of the list (`head1`) involves creating a new instance of `struct1` (`creates1`). The values to fill it are taken from the first element of the list, except for the value of the field `next`, which is empty (`createv1b` creates an empty `variant1`). Getting the list without its first element only involves indexing the `next` field of the first element of `list1`.

```
let is_empty1 = proc(l:list1 -> bool) ; vis1b(l)

let head1 = proc(l:list1 -> struct1)
 creates1(index1-a(vind1a(l)),index1-b(vind1a(l)),createv1b())

let tail1 = proc(l:list1 -> list1) ; index1-c(vind1a(l))
```

Result of Stage One

The result of substituting the newly identified `list1`, `head1`, `is_empty1`, and `tail1` in the printing operation can be seen in figure A–4.

```
rec let print_board = proc(board:list1)
 if ~is_empty1(board) do
 begin
  writeString("BOARD ")
  let AA := head1(board)
  writeInt(index1-a(AA))
  writeString(index1-b(AA)) ; writeString("'n")
  print_board(tail1(board))
 end
```

Figure A–4: Print operation after stage one of the structure level.

A.3.2 Stage Two

At this stage more complex operations on `list1` are identified. In this case, only the printing operation is identified on it. In general, if a data structure

contains substructures, identifying complex operations on it involves identifying complex operations on its substructures first. In this case, in order to define a print operation on a list, a print operation needs to be defined on each element of the list first.

Intermediate Definitions at Stage Two

The only operation on `struct1` identified as part of `list1` in this example is `print_struct1`, which is defined as printing the string "**BOARD**", and the position and color fields of the structure.

```
let print_struct1 = proc(s:struct1)
begin
 writeString("BOARD ")
 writeInt(index1-a(s))
 writeString(index1-b(s))
 writeString("'n")
end
```

Intermediate Result of Stage Two

Figure A–5 shows the resulting form of `print_board` after the substitution of the operation `print_struct1`.

```
rec let print_board = proc(board:list1)
 if ~is_empty1(board) do
 begin
  print_struct1(head1(board))
  print_board(tail1(board))
 end
```

Figure A–5: Intermediate print operation at stage two of the structure level.

More Definitions at Stage Two

Now it is very easy to understand the printing operation on `list1` as involving the printing of each element of the list, and getting the rest of the list until the list is empty. Thus, this is how `print_list1` is defined:

```
rec let print_list1 = proc(l:list1)
if ~is_empty1(l) do
begin
 print_struct1(head1(l))
 print_list1(tail1(l))
end
```

Result of Stage Two and Structure Level

Once `print_list1` is defined and made persistent (in the same way as all the other operations) it disappears from the program after the second stage. What is left in the program is a call to it when it is used:

`print_list1(X)` .

This call, as well as the calls to the other persistent operations, require the declaration of their use from the persistent store. I haven't introduced these declarations for clarity, but an example is given at the end of the explanation on the structure level, section 6.1.1. Figure A–6 shows how it looks in the case of `print_list1`.

```
use persistent store with print_list1:proc(list1) in
 print_list1(X)
```

Figure A–6: Use of print operation in the program after structure level.

The definition of `print_list1` used in the program has to be the same as the one implemented in the persistent store. This is shown by the need to declare the type of `print_list1`.

Appendix B

Example of Analysis Document

This is an extract of the record for the used-defined type `struct1`. It is taken from one of the experiments I performed, the *Mu Torere* experiment (described in section 5.3) and it corresponds to the second stage of the structure level.

The name of the user-defined type identified is `struct1`. The name does not change from the one given at the language level (which reflects the specialization of the language type `structure`). Two operations for `struct1` are identified in this stage: `print_struct1` and `modify_struct1`. Here I will illustrate the information recorded for an operation with the operation `print_struct1`.

The name of the operation is `print_struct1`, and it does not have an **OLD NAME** because it was not implemented as a procedure in the original program. **BODY** is the implementation of the operation, and **CALL** indicates the way in which the operation is to be used in the program, together with the type of its arguments (one in this case). A chunk of code that corresponds to the body of the operation, was found in the module of the program I call the printing module (where all the print operations performed in the program are kept). In particular it was found as part of a procedure called `print_board`. An **ARBITRARY DECISION** taken during substitution is also recorded, together with the actual **SUBSTITUTION** performed.

NEW NAME = struct1
OLD NAME = struct1
OPERATIONS = print_struct1, modify_struct1

INFORMATION OF OPERATION:
OPERATION = print_struct1 (no old name)
BODY =

```
begin
 writeString("BOARD ")
 writeInt(index1-a(s))
 writeString(index1-b(s))
 writeString("'n")
end
```

CALL = print_struct1(<var:struct1>)
WHERE OPERATION
MODULE = printing
CHUNK1 = print_board
PATTERNS OF BODY =

```
writeString("BOARD ")
writeInt(index1-a(AA))
writeString(index1-b(AA))
writeString("'n")
```

ARBITRARY DECISIONS = delete let AA := head1(board) and make the call to print_struct1 with head1(board) directly (instead of doing it with AA).
SUBSTITUTION =

```
writeString("BOARD ")
writeInt(index1-a(AA))
writeString(index1-b(AA))
writeString("'n")
```

is substituted for:

```
print_struct1(head1(board))
```

Figure B–1: Example record in the analysis document.

Appendix C

Some Napier88 Syntax

In this Appendix, I present some definitions and examples of the Napier88 syntax. They are based on the Napier88 manual [Morrison *et al* 89], and are meant to cover only elements of the language mentioned in this thesis. For a more complete description of the language see [Morrison *et al* 89].

C.1 Structures

For identifiers I_1,...,I_n and types t_1,...,t_n, `structure`(I_1:t_1;...;I_n:t_n) is the type of a structure with fields I_i and corresponding types t_i, for i = 1..n. In other words, objects of different types can be grouped together into a structure, and the fields of a structure have identifiers that are unique within that structure.

C.1.1 Creation of Structures

A structure can be created in two ways. It can be created with a type identifier:

```
type struct1 is structure(a:integer ; b:string) .
```

This declares a structure class `struct1` with two fields of type `integer` and `string`. An instance of `struct1` can be created in the following way:

```
let s1 = struct1(3,"example") ,
```

where the field `a` has as value `3`, and `b` has as value `"example"`.

A structure can also be created without a type identifier:

```
let s1 = struct(a:=3 ; b:="example") .
```

C.1.2 Indexing and Assignment

To obtain a field of a structure, the field identifier is used as an index. For example, `s1(a)` yields 3. A new value can be assigned to a field of a structure by the simple mechanism of indexing the field, and assigning a new value to it: `s1(a) := 1 .`

C.2 Variants

For identifiers I_1,...,I_n and types t_1,...,t_n, `variant`(I_1:t_1;...;I_n:t_n) is the type of a variant with identifiers I_i and corresponding types t_i, for i = 1..n. For example,

```
type variant1 is variant(cons:struct1 ; empty:null) ,
```

declares a type `variant1` whose instances will have a value either in the field `cons` (of type `struct1`) or in the field `empty` (of type `null`), but only in one of them, i.e., an instance cannot have values in both fields at the same time.

C.2.1 Variant Values and is and isnt

A variant value may be formed by naming the variant type and injecting the identifier-value pair into it. For example,

```
let v1 = variant1(cons:struct1(3,"example")) ,
```

declares a variant `v1` of type `variant1` injected with the value `struct1(3,"example")` in its identifier `cons`.

A variant can be tested for having a particular identifier. For example, `v1 is cons` is legal and yields `true`, as does `v1 isnt empty`, whereas `v1 is empty` yields `false`.

C.2.2 Variant Usage

The value of a variant may be projected using the quote (') notation. For example, `v1'cons` yields `struct1(3,"example")`.

To assign to a variant the full syntax must be used:

```
v1 = variant1(cons:struct1(1,"example")) .
```

C.2.3 Projection of Variants

To facilitate static type checking, and prevent side effects, a value injected into a variant is rebound to a constant location by the `project` clause. For example, in

```
project v1 as V1 onto
 cons: V1(a)
 default: 1
```

the projected value `v1` is given a constant binding to `V1`, the scope of which is the clause on the right hand side of the colons. For projection, the variant is compared to each of the labels on the left-hand side of the colons (`cons` and `default`). The first match causes the corresponding clause on the right-hand side to be executed.

Without the project clause the previous operation would be expressed in the following way:

```
if v1 is cons then v1'cons(a)
else 1 .
```

C.3 Lists from Structures and Variants

A list in Napier88 is created as a recursive type (`rec`) using `variant` and `structure`. For example,

```
rec list1 is variant(cons:struct2 ; empty:null)
& struct2 is structure(a:integer ; b:string ; c:list1)
```

defines a list that can contain no elements (when the variant has the identifier `empty` injected), or an infinite number of them by virtue of the recursion (`list1` points to `struct2`, which in turn points to `list1`). All the elements of `list1` are of type `struct2`.

C.4 Procedures

Procedures in Napier88 are abstractions over expressions, if they return a value, and clauses of type void if they do not. For any data types t_1,...,t_n and t, `proc`(t_1;...;t_n → t) is the type of a procedure with parameter types t_i, for i=1..n and result type t. The type of a resultless procedure is `proc`(t_1;...;t_n). For example,

```
let id = proc(x:int → int) ; x ,
```

declares a procedure that is the identity procedure for all integers, i.e., given an integer, the procedure `id` will return the same integer.

C.5 Polymorphism and Parameterised Types

Polymorphism permits abstraction over type. For any procedure type, `proc`(t_1;...;t_n → t) and type identifiers T_1,...,T_m, `proc`[T_1;...;T_m](t_1;...;t_n → t) is the type `proc`(t_1;...;t_n → t) universally quantified by types T_1,...,T_m. For example, the polymorphic procedure:

```
let id = proc[t](x:t → t) ; x
```

declares a procedure that is the identity procedure for all types, that is, an infinite number of identity procedures. The square brackets signify that the procedure type is universally quantified by a type `t`, and once given that type, the procedure is from type `t` to `t`. When this polymorphic definition is used, a particular type is injected, e.g. `id[int](3)` yields `3`.

Parameterised types can also be defined, and they are useful when used with recursive variants. For example, a generic type for lists might be:

```
rec type list[t] is variant(cons:node[t] ; empty:null)
& node[s] is structure(hd:s ; tl:list[s]) .
```

C.6 Abstract Data Types

For any type identifiers W_1,...,W_m, identifiers I_1,...,I_n, and types t_1,...,t_n, `abstype`[W_1;...;W_m](I_1:t_1;...;I_n:t_n), is the type of an existentially quantified data type. These are abstract data types, and they can be used where the data object displays some abstract behaviour independent of representation type. For example,

```
type int_stack is abstype[Stack](create_stack:Stack;
                                 push:proc(int,Stack → Stack))
```

declares the type `int_stack` as abstract. The type identifier in the square brackets (`Stack`) is the witness type identifier, and is the type that is abstracted over. The abstract data type interface is declared between the round brackets. In this case,

the type only has two elements: a field `create_stack` with type `Stack`, and a procedure `push` with type `proc(int,Stack → Stack)`.

C.6.1 Creation and Use of Abstract Data Objects

An instance of type `int_stack` can be created by declaring an abstract object `Int_stack`. Following from the type definition `int_stack`, `Int_stack` contains the concrete (as opposed to abstract) witness type `list`, an empty list as the value of the field `create_stack`, and the implemented procedure corresponding to the field `push`. In the creation, the values must be in one-one type correspondence with the type definition.

```
let Int_stack = int_stack[list](list(empty:nil),
                                proc(a:int ; s:list → list)
                                 list(cons:struct(hd = a ; tl = s))) ,
```

where the type of `list` is:

```
rec type list is variant(cons:node; empty:null)
 & node is structure(hd:int ; tl:list) .
```

In Napier88, the `use` clause is introduced to define a constant binding for the object. This binding can then be indexed to refer to the values in a manner that is statically checkable. For example,

```
use Int_stack as p in p(push(3,p(create_stack)))
```

applies the procedure `push` to the value `create_stack` for the abstract data object `Int_stack`, which is bound to the constant `p`.

C.7 Persistence

The root of persistence is obtained by calling the predefined procedure `ps` which has the type: `proc( → env)`, where `env` is the type of environments in Napier88. Thus, the root of the persistent store is of type environment, and when a program is activated, the root contains all the standard bindings provided by the system.

One of them is the procedure to create environments: `environment`, which has the same type as `ps`, and another is the environment `User` where the user of the system can make objects persistent. The example below uses the procedure `environment` to create two new environments: `stack`, and `generic_stack_env`, e.g., `let stack = environment()`. These enviroments are then made persistent, `stack` in `User` and `generic_stack_env` in `stack`. The procedure `cons` is made persistent in `generic_stack_env`.

```
use ps with environment:proc(-> env);
            User:env in
begin
 in User let stack = environment()
 use User with stack:env in
 begin
  let generic_stack_env = environment()
  in generic_stack_env let cons = cons
  in stack let generic_stack_env = generic_stack_env
 end
end
```

Appendix D

The Maze Program

D.1 Base Program

The program contains a declaration of data types, persistent objects, global variables, procedures, and the main program. There are two important types of lists: one to implement the type of the maze, `list1`, and another to implement the type of the sequence of neighbours of a cell, and the type of a path, `list2`. The persistent object declaration contains some standard I/O operations which Napier88 keeps in the standard persistent environment. There are three global variables, one for the maze, `maze`, another for the path, `total`, and a `boolean` variable to indicate if a path has been found, `success`. Amongst the procedures, `create_maze` creates the maze with data kept in the procedure; `find_path` is a recursive procedure that returns a path between two given cells (the path is empty if there isn't one); `find_path` calls to `find_neigh` to get the neighbours of a cell in the maze, and it keeps track of the paths being explored at each moment in the variable `path` that is also fed to `find_path`. A path is built backwards in `find_path`, so it has to be reversed (`reverse`) before it is printed (`print_path`). Other auxiliary procedures present in the program are: `append`, `member`, and `print_maze`. The main program first calls `create_maze`, asks the user for the two cells to find a path between, calls `find_path` to find a path, and if a path is found reverses and prints it,if the path is empty, an appropriate message is printed instead.

```
!DECLARATION OF PERSISTENT OBJECTS
let ps = PS()
use ps with IO:env in
use IO with writeString:proc(string) ; writeInt:proc(int) ;
readInt:proc( -> int) in
begin
!DECLARATION OF TYPES
rec type list2 is variant(cons: node2 ; empty:null)
&       node2 is structure(key:int ; next:list2)

rec type list1 is variant(cons:node1 ; empty:null)
&       node1 is structure(key:int ; neigh:list2 ; next:list1)

!DECLARATION OF GLOBAL VARIABLES
let maze := list1(empty:nil)   !maze
let total := list2(empty:nil)  !final path between two cells
let success := false           !boolean to indicate a path found

!DECLARATION OF PROCEDURES
!CREATE_MAZE CREATES THE MAZE INCLUDING THE DATA
let create_maze = proc()
begin
maze :=
list1(cons:node1(22,list2(cons:node2(23,list2(cons:node2(32,list2(empty:nil))))),
list1(cons:node1(23,list2(cons:node2(22,list2(cons:node2(24,list2(empty:nil))))),
list1(cons:node1(24,list2(cons:node2(23,list2(cons:node2(25,list2(empty:nil))))),
list1(cons:node1(25,list2(cons:node2(24,list2(cons:node2(26,list2(empty:nil))))),
list1(cons:node1(26,list2(cons:node2(25,list2(cons:node2(27,list2(empty:nil))))),
list1(cons:node1(27,list2(cons:node2(26,list2(cons:node2(28,list2(empty:nil))))),
list1(cons:node1(28,list2(cons:node2(27,list2(cons:node2(38,list2(empty:nil))))),
list1(cons:node1(210,list2(cons:node2(211,list2(empty:nil))),
list1(cons:node1(211,list2(cons:node2(210,list2(cons:node2(212,list2(empty:nil))))),
list1(cons:node1(212,list2(cons:node2(211,list2(cons:node2(312,list2(empty:nil))))),
list1(cons:node1(32,list2(cons:node2(22,list2(cons:node2(42,list2(empty:nil))))),
list1(cons:node1(38,list2(cons:node2(28,list2(cons:node2(48,list2(empty:nil))))),
list1(cons:node1(312,list2(cons:node2(212,list2(cons:node2(412,list2(empty:nil))))),
list1(cons:node1(42,list2(cons:node2(32,list2(cons:node2(52,
 list2(cons:node2(43,list2(empty:nil)))))))),
list1(cons:node1(43,list2(cons:node2(42,list2(cons:node2(44,list2(empty:nil))))),
list1(cons:node1(44,list2(cons:node2(43,list2(cons:node2(45,list2(empty:nil))))),
list1(cons:node1(45,list2(cons:node2(44,list2(cons:node2(46,list2(empty:nil))))),
list1(cons:node1(46,list2(cons:node2(45,list2(cons:node2(56,list2(empty:nil))))),
list1(cons:node1(48,list2(cons:node2(38,list2(cons:node2(58,
 list2(cons:node2(49,list2(empty:nil)))))))),
list1(cons:node1(49,list2(cons:node2(48,list2(cons:node2(410,list2(empty:nil))))),
list1(cons:node1(410,list2(cons:node2(49,list2(cons:node2(411,list2(empty:nil))))),
list1(cons:node1(411,list2(cons:node2(410,list2(cons:node2(412,list2(empty:nil))))),
list1(cons:node1(412,list2(cons:node2(411,list2(cons:node2(312,list2(empty:nil))))),
list1(cons:node1(52,list2(cons:node2(42,list2(cons:node2(62,list2(empty:nil))))),
list1(cons:node1(56,list2(cons:node2(46,list2(cons:node2(66,list2(empty:nil))))),
list1(cons:node1(58,list2(cons:node2(48,list2(cons:node2(68,list2(empty:nil))))),
list1(cons:node1(62,list2(cons:node2(52,list2(cons:node2(63,list2(empty:nil))))),
list1(cons:node1(63,list2(cons:node2(62,list2(cons:node2(64,list2(empty:nil))))),
list1(cons:node1(64,list2(cons:node2(63,list2(cons:node2(65,
 list2(cons:node2(74,list2(empty:nil)))))))),
list1(cons:node1(65,list2(cons:node2(64,list2(cons:node2(66,list2(empty:nil))))),
list1(cons:node1(66,list2(cons:node2(65,list2(cons:node2(56,list2(empty:nil))))),
list1(cons:node1(68,list2(cons:node2(58,list2(cons:node2(69,list2(empty:nil))))),
list1(cons:node1(69,list2(cons:node2(68,list2(cons:node2(610,list2(empty:nil))))),
```

```
list1(cons:node1(610,list2(cons:node2(69,list2(cons:node2(611,
 list2(cons:node2(710,list2(empty:nil)))))))),
list1(cons:node1(611,list2(cons:node2(610,list2(cons:node2(612,list2(empty:nil))))))),
list1(cons:node1(612,list2(cons:node2(611,list2(cons:node2(712,list2(empty:nil))))))),
list1(cons:node1(74,list2(cons:node2(64,list2(cons:node2(84,list2(empty:nil))))))),
list1(cons:node1(710,list2(cons:node2(610,list2(cons:node2(810,list2(empty:nil))))))),
list1(cons:node1(712,list2(cons:node2(612,list2(cons:node2(812,list2(empty:nil))))))),
list1(cons:node1(82,list2(cons:node2(92,list2(empty:nil)))),
list1(cons:node1(84,list2(cons:node2(85,list2(cons:node2(74,list2(empty:nil))))))),
list1(cons:node1(85,list2(cons:node2(84,list2(cons:node2(86,list2(empty:nil))))))),
list1(cons:node1(86,list2(cons:node2(85,list2(cons:node2(87,
 list2(cons:node2(96,list2(empty:nil)))))))),
list1(cons:node1(87,list2(cons:node2(86,list2(cons:node2(88,list2(empty:nil))))))),
list1(cons:node1(88,list2(cons:node2(87,list2(empty:nil)))),
list1(cons:node1(810,list2(cons:node2(710,list2(cons:node2(910,list2(empty:nil))))))),
list1(cons:node1(812,list2(cons:node2(712,list2(empty:nil)))),
list1(cons:node1(92,list2(cons:node2(82,list2(cons:node2(102,list2(empty:nil))))))),
list1(cons:node1(96,list2(cons:node2(86,list2(cons:node2(106,list2(empty:nil))))))),
list1(cons:node1(910,list2(cons:node2(810,list2(cons:node2(1010,list2(empty:nil))))))),
list1(cons:node1(102,list2(cons:node2(92,list2(cons:node2(103,list2(empty:nil))))))),
list1(cons:node1(103,list2(cons:node2(102,list2(cons:node2(104,list2(empty:nil))))))),
list1(cons:node1(104,list2(cons:node2(103,list2(cons:node2(114,list2(empty:nil))))))),
list1(cons:node1(106,list2(cons:node2(96,list2(cons:node2(116,list2(empty:nil))))))),
list1(cons:node1(108,list2(cons:node2(109,list2(cons:node2(118,list2(empty:nil))))))),
list1(cons:node1(109,list2(cons:node2(108,list2(cons:node2(1010,list2(empty:nil))))))),
list1(cons:node1(1010,list2(cons:node2(109,list2(cons:node2(1011,
 list2(cons:node2(910,list2(empty:nil)))))))),
list1(cons:node1(1011,list2(cons:node2(1010,list2(cons:node2(1012,list2(empty:nil))))))),
list1(cons:node1(1012,list2(cons:node2(1011,list2(empty:nil)))),
list1(cons:node1(114,list2(cons:node2(104,list2(cons:node2(124,list2(empty:nil))))))),
list1(cons:node1(116,list2(cons:node2(106,list2(cons:node2(126,list2(empty:nil))))))),
list1(cons:node1(118,list2(cons:node2(108,list2(cons:node2(128,list2(empty:nil))))))),
list1(cons:node1(122,list2(cons:node2(123,list2(empty:nil)))),
list1(cons:node1(123,list2(cons:node2(122,list2(cons:node2(124,list2(empty:nil))))))),
list1(cons:node1(124,list2(cons:node2(123,list2(cons:node2(125,
 list2(cons:node2(114,list2(empty:nil)))))))),
list1(cons:node1(125,list2(cons:node2(124,list2(cons:node2(126,list2(empty:nil))))))),
list1(cons:node1(126,list2(cons:node2(125,list2(cons:node2(116,list2(empty:nil))))))),
list1(cons:node1(128,list2(cons:node2(118,list2(cons:node2(129,list2(empty:nil))))))),
list1(cons:node1(129,list2(cons:node2(128,list2(cons:node2(1210,list2(empty:nil))))))),
list1(cons:node1(1210,list2(cons:node2(129,list2(cons:node2(1211,list2(empty:nil))))))),
list1(cons:node1(1211,list2(cons:node2(1210,list2(cons:node2(1212,list2(empty:nil))))))),
list1(cons:node1(1212,list2(cons:node2(1211,list2(empty:nil)))),
list1(empty:nil))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
end

!PRINT_MAZE PRINTS OUT THE LIST THAT REPRESENTS THE MAZE.
rec let print_maze = proc(maze:list1)
if maze isnt empty do
begin
 writeInt(maze'cons(key)) ; writeString("'n")
 let x := maze'cons(neigh)
 while x isnt empty do
 begin
  writeInt(x'cons(key))
  x := x'cons(next)
 end
 writeString("'n")
```

```
 print_maze(maze'cons(next))
end

!PRINT_PATH PRINTS THE LIST THAT REPRESENTS A PATH.
rec let print_path = proc(l:list2)
if l isnt empty do
begin
 writeInt(l'cons(key))
 print_path(l'cons(next))
end

!REVERSE REVERSES A LIST
let reverse = proc(l:list2 -> list2)
begin
 let temp := list2(empty:nil)
 while l isnt empty do
 begin
  temp := list2(cons:struct(key=l'cons(key) ; next=temp))
  l := l'cons(next)
 end
 temp
end

!MEMBER LOOKS IF AN ELEMENT IS A MEMBER OF A LIST
rec let member = proc(elem:int ; l:list2 -> bool)
if l is empty then false
else
 if l'cons(key) = elem then true
 else member(elem,l'cons(next))

!APPEND ADDS AN ELEMENT AT THE END OF A LIST
rec let append = proc(l:list2 ; last:int -> list2)
begin
 if l is empty then l := list2(cons:struct(key=last ; next=l))
 else l'cons(next) := append(l'cons(next),last)
 l
end

!FIND_NEIGH LOOKS FOR A CELL IN THE LIST THAT REPRESENTS THE MAZE.
!IF IT FINDS IT, IT RETURNS THE SUBLIST OF NEIGHBOURS OF
!THAT ELEMENT. OTHERWISE, IT RETURNS AN EMPTY LIST.
rec let find_neigh = proc(x:int ; maze:list1 -> list2)
begin
 let elem := maze
 let exit := false
 let neighbours := list2(empty:nil)
 while elem isnt empty  and ~exit do
  if elem'cons(key) ~= x then elem := elem'cons(next)
  else exit := true
  if exit do if elem'cons(key) = x do neighbours := elem'cons(neigh)
  neighbours
end

!FIND_PATH FINDS A PATH IN THE MAZE BETWEEN TWO GIVEN CELLS.
rec let find_path = proc(x,y:int ; path:list2 -> list2)
begin
 if x = y then
 begin                      !A path has been found and the cell is added
  total := append(total,x)  !to the global variable for the path.
```

```
  success := true
 end
 else
  if ~member(x,path) do     !checking that there are not repeated cells.
  begin
   let subx := find_neigh(x,maze)         !get neighbours of a cell.
   while ~success and subx isnt empty do  !depth-first search until
   begin                                  !a path is found or
    let z := subx'cons(key)               !the maze is exhausted.
    subx := subx'cons(next)
    total := find_path(z,y,append(path,x))!recursive call
   end                                    !with each neighbour.
   if success do total := append(total,x)
  end
total
end

!MAIN PROGRAM
create_maze()      !print_maze(maze)
writeString( !GET INITIAL AND FINAL CELLS OF THE PATH FROM USER.
"Indicate the number of the starting cell of the path. CELL: ")
let x := readInt()
writeString("
Indicate the number of the ending cell of the path. CELL: ")
let y := readInt()
total := find_path(x,y,total)                    !find a path.
if total is empty then writeString("There is no path.")
else
begin                            !reverse and print the path.
 total := reverse(total)
 writeString("The path is: 'n")
 print_path(total)
end
end
```

D.2 Program at Language Level

The program is part in the original file, and part in the persistent store. I show the part left in the original file. In the declaration of persistent objects we can see the operations that are used from the persistent store.

```
!DECLARATION OF TYPES
rec type variant2 is variant(cons: node2 ; empty:null)
&        node2 is structure(key:int ; next:variant2)
rec type variant1 is variant(cons:node1 ; empty:null)
&        node1 is structure(key:int ; neigh:variant2 ; next:variant1)
!DECLARATION OF PERSISTENT OBJECTS
let ps = PS()
use ps with User,IO:env ; environment:proc( -> env) in
use IO with PrimitiveIO:env ; writeString:proc(string) ; writeInt:proc(int) ;
            readInt:proc( -> int) ; endOfInput:proc( -> bool) ; makeReadEnv: proc (file -> env) in
use PrimitiveIO with open:proc(string,int -> file) ; close:proc(file -> int) in
use User with maze_env:env in
use maze_env with struct_var_env:env in
use struct_var_env with create_node1:proc(int,variant2,variant1 -> node1) ;
                        head1:proc(node1 -> int) ;
                        tail1:proc(node1 -> variant1) ;
                        neighs:proc(node1 -> variant2) ;
                        create_node2:proc(int,variant2 -> node2) ;
                        head2:proc(node2 -> int) ;
                        tail2:proc(node2 -> variant2) ;
                        create_variant1:proc( -> variant1) ;
                        empty1:proc(variant1 -> bool) ;
                        value1:proc(node1 -> variant1) ;
                        usage1:proc(variant1 -> node1) ;
                        create_variant2:proc( -> variant2) ;
                        empty2:proc(variant2 -> bool) ;
                        value2:proc(node2 -> variant2) ;
                        usage2:proc(variant2 -> node2) in
begin
!DECLARATION OF GLOBAL VARIABLES
let maze := create_variant1()   !variable for the maze
let total := create_variant2()  !variable for the final path between two cells
let success := false            !boolean to indicate if a path has been found

!DECLARATION OF PROCEDURES
let reverse = proc(l:variant2 -> variant2) !REVERSE REVERSES A LIST
begin
 let temp := create_variant2()
 while ~empty2(l) do
 begin
  temp := value2(create_node2(head2(usage2(l)),temp))
  l := tail2(usage2(l))
 end
 temp
end

!MEMBER LOOKS IF AN ELEMENT IS A MEMBER OF A LIST
rec let member = proc(elem:int ; l:variant2 -> bool)
if empty2(l) then false
else
 if head2(usage2(l)) = elem then true
 else member(elem,tail2(usage2(l)))

!APPEND ADDS AN ELEMENT AT THE END OF A LIST
rec let append = proc(l:variant2 ; last:int -> variant2)
begin
 if empty2(l) then l := value2(create_node2(last,l))
 else
  l := value2(create_node2(head2(usage2(l)),append(tail2(usage2(l)),last)))
```

```
 1
end

!PRINT_MAZE PRINTS OUT A LIST THAT REPRESENTS THE MAZE
rec let print_maze = proc(maze:variant1)
if ~empty1(maze) do
begin
 writeInt(head1(usage1(maze))) ; writeString("'n")
 let x := neighs(usage1(maze))
 while ~empty2(x) do
 begin
  writeInt(head2(usage2(x)))
  x := tail2(usage2(x))
 end
 writeString("'n")
 print_maze(tail1(usage1(maze)))
end

!PRINT_PATH PRINTS A LIST THAT REPRESENTS A PATH BETWEEN TWO CELLS
rec let print_path = proc(l:variant2)
if ~empty2(l) do
begin
 writeInt(head2(usage2(l)))
 print_path(tail2(usage2(l)))
end

!CREATE_MAZE CREATES THE REPRESENTATION OF THE MAZE READING THE DATA
!FROM THE FILE DATA_MAZE
let create_maze = proc()
begin
!declaration of local persistent objects
 let fichero = open("data_maze",0)
 let readfile = makeReadEnv(fichero)
 use readfile with readInt:proc( -> int) ; endOfInput:proc( -> bool) in
 begin
 !local variables declaration
  let neighbours := create_variant2()
  let num := 0
!local procedure add_cell_in_maze declaration
   rec let add_cell_in_maze = proc(cell:int ; neighbours:variant2 ; maze:variant1 -> variant1)
  begin
   if empty1(maze) then maze := value1(create_node1(cell,neighbours,maze))
   else
    maze := value1(create_node1(head1(usage1(maze)),neighs(usage1(maze)),
                                add_cell_in_maze(cell,neighbours,tail1(usage1(maze)))))
   maze
  end
!MAIN BODY OF THE PROCEDURE CREATE_MAZE
  while ~endOfInput() do
  begin
   num := readInt()
   neighbours := create_variant2()
   while num ~= 0 do
   begin
    neighbours := append(neighbours,num)
    num := readInt()
   end
   maze := add_cell_in_maze(head2(usage2(neighbours)),tail2(usage2(neighbours)),maze)
  end
```

```
 end
let aa = close(fichero)
if aa ~= 0 do writeString("error when closing the data file")
end

!FIND_NEIGH LOOKS FOR AN ELEMENT IN A LIST (THAT IS SUPPOSSED TO BE A CELL
!IN THE MAZE) AND IF IT FINDS IT, IT GIVES BACK THE SUBLIST CORRESPONDING TO
!THE NEIGHBOURS OF THAT ELEMENT. OTHERWISE, IT GIVES BACK THE EMPTY LIST.
rec let find_neigh = proc(x:int ; maze:variant1 -> variant2)
begin
 let elem := maze
 let exit := false
 let neighbours := create_variant2()
 while ~empty1(elem)  and ~exit do
  if head1(usage1(elem)) ~= x then elem := tail1(usage1(elem))
  else exit := true
  if exit do if head1(usage1(elem)) = x do neighbours := neighs(usage1(elem))
  neighbours
end

!FIND_PATH FINDS A PATH BETWEEN TWO CELLS OF THE MAZE
rec let find_path = proc(x,y:int ; path:variant2 -> variant2)
begin
 if x = y then
 begin
  total := append(total,x)
  success := true
 end
 else
  if ~member(x,path) do
  begin
   let subx := find_neigh(x,maze)
   while ~success and ~empty2(subx) do
   begin
    let z := head2(usage2(subx))
    subx := tail2(usage2(subx))
    total := find_path(z,y,append(path,x))
   end
   if success do total := append(total,x)
  end
total
end

!MAIN PROGRAM
create_maze()      !print_maze(maze)
writeString("Indicate the number of the starting cell of the path. CELL: ")
let x := readInt()
writeString("Indicate the number of the ending cell of the path. CELL: ")
let y := readInt()
total := find_path(x,y,total)
if empty2(total) then writeString("There is no path.")
else
begin
 total := reverse(total)
 writeString("The path is: 'n")
 print_path(total)
end
end
```

D.3 Program at Structure Level

I show the part of the program in the original file. In the declaration of persistent objects we can see the operations that are used from the persistent store (identified at the structure level and which are implemented in terms of operations at the language level).

```
!DECLARATION OF TYPES
rec type variant2 is variant(cons: node2 ; empty:null)
&        node2 is structure(key:int ; next:variant2)
rec type variant1 is variant(cons:node1 ; empty:null)
&        node1 is structure(key:int ; neigh:variant2 ; next:variant1)

type list1 is variant1
type list2 is variant2

!DECLARATION OF PERSISTENT OBJECTS
let ps = PS()
use ps with User,IO:env in
use IO with PrimitiveIO:env ;
            writeString:proc(string) ; writeInt:proc(int) ;
            readInt:proc( -> int) ; endOfInput:proc( -> bool) ;
            makeReadEnv: proc (file -> env) in
use PrimitiveIO with open:proc(string,int -> file) ;
                     close:proc(file -> int) in
use User with maze_env:env in
use maze_env with list_file_env:env in
use list_file_env with create_list1:proc( -> list1) ;
                  empty_list1:proc(list1 -> bool) ;
                  head_list1:proc(list1 -> int) ;
                  sub_list1:proc(list1 -> list2) ;
                  tail_list1:proc(list1 -> list1) ;
                  append_list1:proc(int,list2,list1 -> list1) ;
                  create_list2:proc( -> list2) ;
                  empty_list2:proc(list2 -> bool) ;
                  head_list2:proc(list2 -> int) ;
                  tail_list2:proc(list2 -> list2) ;
                  member_list2:proc(int,list2 -> bool) ;
                  append_list2:proc(int,list2 -> list2) ;
                  reverse_list2:proc(list2 -> list2) ;
                  print_list2:proc(list2) in
begin

!DECLARATION OF GLOBAL VARIABLES
let maze := create_list1()   !variable for the maze
let total := create_list2()  !variable for the final path between two cells
let success := false           !boolean to indicate if a path has been found

!DECLARATION OF PROCEDURES
!CREATE_MAZE CREATES THE REPRESENTATION OF THE MAZE READING THE DATA
!FROM THE FILE DATA_MAZE
let create_maze = proc()
begin                    !LOCAL PROCEDURE DECLARATION
```

```
!open_file opens the file containing the data for the maze
let open_file = proc( -> file)
begin
 let fichero = open("data_maze",0)
 fichero
end
!close_file closes the file containing the data for the maze
let close_file = proc(fichero:file)
begin
 let aa = close(fichero)
 if aa ~= 0 do writeString("error when closing the data file")
end
!declaration of local persistent objects
let fichero = open_file()
let readfile = makeReadEnv(fichero)
use readfile with readInt:proc( -> int) ; endOfInput:proc( -> bool) in
begin
!read_line reads a line of data from the file. each line contains
!a cell and its neighbours
let read_line = proc( -> list2)
begin
 let num := readInt()
 let l := create_list2()
 while num ~= 0 do
 begin
 l := append_list2(num,l)
 num := readInt()
 end
 l
end
!local variables declaration
 let neighbours := create_list2()
 while ~endOfInput() do        !main body of the procedure create_maze
 begin
  neighbours := read_line()
  maze := append_list1(head_list2(neighbours),tail_list2(neighbours),maze)
 end
end
 close_file(fichero)
end

!FIND_NEIGH LOOKS FOR AN ELEMENT IN A LIST (THAT IS SUPPOSSED TO BE A CELL
!IN THE MAZE) AND IF IT FINDS IT, IT GIVES BACK THE SUBLIST CORRESPONDING TO
!THE NEIGHBOURS OF THAT ELEMENT. OTHERWISE, IT GIVES BACK THE EMPTY LIST.
rec let find_neigh = proc(x:int ; maze:list1 -> list2)
begin
 let elem := maze
 let exit := false
 let neighbours := create_list2()
 while ~empty_list1(elem)  and ~exit do
  if head_list1(elem) ~= x then elem := tail_list1(elem)
  else exit := true
  if exit do if head_list1(elem) = x do neighbours := sub_list1(elem)
  neighbours
end


!FIND_PATH FINDS A PATH BETWEEN TWO CELLS OF THE MAZE
rec let find_path = proc(x,y:int ; path:list2 -> list2)
```

```
begin
 if x = y then
 begin
  total := append_list2(x,total)
  success := true
 end
 else
  if ~member_list2(x,path) do
  begin
   let subx := find_neigh(x,maze)
   while ~success and ~empty_list2(subx) do
   begin
    let z := head_list2(subx)
    subx := tail_list2(subx)
    total := find_path(z,y,append_list2(x,path))
   end
   if success do total := append_list2(x,total)
  end
total
end

!MAIN PROGRAM
create_maze()      !print_list1(maze)
writeString("Indicate the number of the starting cell of the path. CELL: ")
let x := readInt()
writeString("Indicate the number of the ending cell of the path. CELL: ")
let y := readInt()
total := find_path(x,y,total)
if empty_list2(total) then writeString("There is no path.")
else
begin
 total := reverse_list2(total)
 writeString("The path is: 'n")
 print_list2(total)
end
end
```

D.4 Final Program

The final program is part in the original file, and part in the persistent store. I first show the part left in the original file, and then the part in the persistent store which contains operations at the language level, structure level, and domain level. Here I only show the operations corresponding to the domain level analysis since they are part of the final program.

D.4.1 Part of the Final Program in Original File

```
!DECLARATION OF TYPES
rec type variant2 is variant(cons: node2 ; empty:null)
&       node2 is structure(key:int ; next:variant2)
 rec type variant1 is variant(cons:node1 ; empty:null)
&       node1 is structure(key:int ; neigh:variant2 ; next:variant1)

type list1 is variant1
type list2 is variant2

type maze is list1
type path is list2
type neighbours is list2
type cell is int

!DECLARATION OF PERSISTENT OBJECTS
let ps = PS()
use ps with User,IO:env in
use IO with writeString:proc(string) ; readInt:proc( -> int) in
use User with maze_env:env in
use maze_env with maze_objects_env:env in
use maze_objects_env with readCell:proc( -> cell) ;
                          initialize_path:proc( -> path) ;
                          empty_path:proc(path -> bool) ;
                          reverse_path:proc(path -> path) ;
                          print_path:proc(path) ;
                          find_path:proc(cell,cell,path,maze -> path) ;
                          empty_maze:proc(maze -> bool) ;
                          create_maze:proc(maze -> maze) ;
                          print_maze:proc(maze) ;
                          empty_maze:proc(maze -> bool) ;
                          create_maze:proc(maze -> maze) ;
                          print_maze:proc(maze) ;
                          success:bool ; total_path:path ;
                          this_maze:maze in
begin !MAIN PROGRAM
!initialize persistent objects, and create maze if it is not created already
success := false
total_path := initialize_path()
if empty_maze(this_maze) do
begin
this_maze := create_maze(this_maze)
print_maze(this_maze)
end
!get initial and final cells of path from user, and find path
writeString("Indicate the number of the starting cell of the path. CELL: ")
let c1 := readCell()
writeString("Indicate the number of the ending cell of the path. CELL: ")
let c2 := readCell()
total_path := find_path(c1,c2,total_path,this_maze)
if empty_path(total_path) then writeString("There is no path.")
else !print the path
begin
 total_path := reverse_path(total_path)
 writeString("The path is: 'n")
 print_path(total_path)
end
end
```

D.4.2 Part of Program in Persistent Store

These are the operations made persistent during the domain level analysis. They make use of operations made persistent during the structure level (see declaration of persistent objects just below). What is shown here is the definition file used to make these operations persistent, and not the content of the persistent store as such. The reason for this is that the version of Napier88 used for these experiments did not have facilities for visualising the persistent store, and it was easier (and clearer) to show the definition file.

We can see that the procedures `find_path` and `create_maze` are modified from their form at the previous levels to solve the problem of making global variables persistent (discussed in section 5.2.4). The solution is to make the global variables persistent, and then to manipulate them as independent persistent objects inside, and outside, the procedures that use them.

```
!DECLARATION OF PERSISTENT OBJECTS
let ps = PS()
use ps with User,IO:env in
use IO with PrimitiveIO:env ;
            writeString:proc(string) ; writeInt:proc(int) ;
            readInt:proc( -> int) ; endOfInput:proc( -> bool) ;
            makeReadEnv: proc (file -> env) in
use PrimitiveIO with open:proc(string,int -> file) ;
                     close:proc(file -> int) in
use User with maze_env:env in
use maze_env with list_file_env,maze_objects_env:env in
use list_file_env with create_list1:proc( -> list1) ;
                  empty_list1:proc(list1 -> bool) ;
                  head_list1:proc(list1 -> int) ;
                  sub_list1:proc(list1 -> list2) ;
                  tail_list1:proc(list1 -> list1) ;
                  append_list1:proc(int,list2,list1 -> list1) ;
                  print_list1:proc(list1) ;
                  create_list2:proc( -> list2) ;
                  empty_list2:proc(list2 -> bool) ;
                  head_list2:proc(list2 -> int) ;
                  tail_list2:proc(list2 -> list2) ;
                  member_list2:proc(int,list2 -> bool) ;
                  append_list2:proc(int,list2 -> list2) ;
                  reverse_list2:proc(list2 -> list2) ;
                  print_list2:proc(list2) in
begin

!DECLARATION OF OPERATIONS ON TYPE CELL
let readCell = proc( -> cell) ; readInt()
let cell_is_in_path = proc(c:cell ; p:path -> bool) ; member_list2(c,p)
let first_cell_in_maze = proc(m:maze -> cell) ; head_list1(m)
let first_cell_in_path = proc(p:path -> cell) ; head_list2(p)
```

```
let first_cell_in_neighbours = proc(n:neighbours -> cell) ; head_list2(n)

!DECLARATION OF OPERATIONS ON TYPE NEIGHBOURS
let initialize_neighbours = proc( -> neighbours) ; create_list2()
let empty_neighbours = proc(n:neighbours -> bool) ; empty_list2(n)
let add_cell_in_neighbours = proc(c:cell ; n:neighbours -> neighbours)
 append_list2(c,n)
let rest_neighbours = proc(n:neighbours -> neighbours) ; tail_list2(n)
let get_neighbours = proc(m:maze -> neighbours) ; sub_list1(m)
!find_neighbours looks for a cell in the maze and if it finds it, it gives back the set
!of neighbours of that cell; otherwise, it gives back an empty set of neighbours.
rec let find_neighbours = proc(c:cell ; m:maze -> neighbours)
 if empty_maze(m) then initialize_neighbours()
 else if first_cell_in_maze(m) = c then get_neighbours(m)
      else find_neighbours(c,rest_of_maze(m))

!DECLARATION OF OPERATIONS ON TYPE PATH
let initialize_path = proc( -> path) ; create_list2()
let empty_path = proc(p:path -> bool) ; empty_list2(p)
let add_cell_in_path = proc(c:cell ; p:path -> path) ; append_list2(c,p)
let print_path = proc(p:path) ; print_list2(p)
let reverse_path = proc(p:path -> path) ; reverse_list2(p)
!find_path finds a path between two cells of the maze
in maze_objects_env let success := false
in maze_objects_env let total_path := initialize_path()
rec let find_path = proc(c1,c2:cell ; try_path:path ; m:maze -> path)
begin
 use maze_objects_env with success:bool ; total_path:path in
 begin
 if c1 = c2 then
 begin
  total_path := add_cell_in_path(c1,total_path)
  success := true
 end
 else
  if ~cell_is_in_path(c1,try_path) do
  begin
   let n := find_neighbours(c1,m)
   while ~success and ~empty_neighbours(n) do
   begin
    let c3 := first_cell_in_neighbours(n)
    n := rest_neighbours(n)
    total_path := find_path(c3,c2,add_cell_in_path(c1,try_path),m)
   end
   if success do total_path := add_cell_in_path(c1,total_path)
  end
total_path
end
end

!DECLARATION OF OPERATIONS ON TYPE MAZE
let initialize_maze = proc( -> maze) ; create_list1()
let empty_maze = proc(m:maze -> bool) ; empty_list1(m)
let print_maze = proc(m:maze) ; print_list1(m)
let add_cell_in_maze = proc(c:cell ; n:neighbours ; m:maze -> maze)
 append_list1(c,n,m)
let rest_of_maze = proc(m:maze -> maze) ; tail_list1(m)

!create_maze creates the representation of an object of type maze
```

```
!reading the data from the file data_maze
let create_maze = proc(m:maze -> maze)
begin
!local procedures declaration
!open_file opens the file containing the data for the maze
let open_file = proc( -> file)
begin
 let fichero = open("data_maze",0)
 fichero
end
!close_file closes the file containing the data for the maze
let close_file = proc(fichero:file)
begin
 let aa = close(fichero)
 if aa ~= 0 do writeString("error when closing the data file")
end
!declaration of local persistent objects
let fichero = open_file()
let readfile = makeReadEnv(fichero)
use readfile with readInt:proc( -> int) ; endOfInput:proc( -> bool) in
begin
!read_line reads a line of data from the file. Each line contains
!a cell and its neighbours
let read_line = proc( -> list2)
begin
 let c := readInt()
 let n := initialize_neighbours()
 while c ~= 0 do
 begin
 n := add_cell_in_neighbours(c,n)
 c := readInt()
 end
 n
end
!main body of the procedure create_maze
 while ~endOfInput() do
 begin
  let n:= read_line()
  m := add_cell_in_maze(first_cell_in_neighbours(n),rest_neighbours(n),m)
 end
end
 close_file(fichero)
 m
end

!DECLARATION OF GLOBAL PERSISTENT VARIABLES
in maze_objects_env let this_maze := initialize_maze()
end
```

Appendix E

The Mu Torere Program

E.1 Prototype One

```
!This program is the final program of the first version for the torere game.
!This program always loses the game because when it has to choose between
!moving adjecent and moving from putahi, it always chooses to move from
!putahi, and to win it needs to move adjacent at certain times. This is
!because it chooses the first rule that accomplishes the board and in
!the rule base the move from putahi is before the move adjacent.
!DECLARATION OF PERSISTENT ENVIRONMENTS
let ps = PS()
use ps with IO,Arithmetical,User:env in
use Arithmetical with abs:proc(int -> int) in
use IO with writeString: proc(string) ; writeInt:proc(int) ;
            writeBool:proc(bool) ;
            readString:proc( -> string) ; readInt:proc( -> int) in
use User with torere_env:env in
use torere_env with log_env:env in
use log_env with games_env:env in
begin

!TYPES OF THE FACT BASE
!color contains the colour the program is playing with:none,white or black
!move indicates whose turn is to move, the program or the user
!start indicates if it is the begining of a game or not: yes,no
!boards is the type of the state of the board in the game
rec type boards is variant(cons:node_boards ; empty:null)
&   node_boards is structure(position:int ; color:string ; next:boards)
!fact_bases is the type of the fact base
type fact_bases is structure(start:string ;
                             move:string ;
                             color:string ;
                             board:boards)

!TYPES OF RULE BASE
!type of pattern1
rec type list1 is variant(cons:node_pattern1 ; empty:null)
&       node_pattern1 is structure(fact_name:string ;
                                   fact_value:string ;
                                   next:list1)

!types for pattern2
type varcons is variant(pos_var:string ; pos_const:int)
rec type list2 is variant(cons:node_pattern2 ; empty:null)
```

```
&       node_pattern2 is structure(position:varcons ;
                                   color:string ;
                                   next:list2)
!type of pattern3
type test is variant(test_id:string ; empty:null)
!rule_bases is the type of the rule base
rec type rule_bases is variant(cons:node_rule ; empty:null)
&       node_rule is structure(rule_id:string ;
                               pattern1:list1 ;
                               pattern2:list2 ;
                               pattern3:test ;
                               right_side:string ;
                               next:rule_bases)

!TYPES OF WORKING MEMORY
!wms is the type of the working memory
type wms is variant(board:boards ;
                    color:string ;
                    move:string ;
                    start:string ;
                    empty:null)

!TYPES OF THE AGENDA
!types for elem1
rec type list3 is variant(cons:node_pos ; empty:null)
&       node_pos is structure(position:int ; next:list3)
type singles is structure(var_pos:string ; pos_list:list3)
!types for elem2
rec type list4 is variant(cons:node_tuples ; empty:null)
&       node_tuples is structure(position1:int ; position2:int ; next:list4)
type tuples is structure(var_pos1:string ; var_pos2:string ; tuples_list:list4)
!types for elem3
rec type list5 is variant(cons:node_triples ; empty:null)
&       node_triples is structure(position1:int ; position2:int ; position3:int ; next:list5)
type triples is structure(var_pos1:string ;
                          var_pos2:string ;
                          var_pos3:string ;
                          triples_list:list5)
!elem_type is the type of the elements of the agenda
type elem_type is variant(elem0:bool ; elem1:singles ; elem2:tuples ; elem3:triples)
!agendas is the type of the input elements for the agenda
rec type agendas is variant(cons:node_agenda ; empty:null)
&       node_agenda is structure(rule_id:string ;
                                 agenda_elem:elem_type ;
                                 next:agendas)

!OTHER TYPES FOR THE PROGRAM
!patterns_results is the type of the element containing all the variable
!results of a rule
rec type patterns_results is variant(cons:node_results ; empty:null)
&   node_results is structure(results:singles ; next:patterns_results)

!argument_pm is the type of the argument to the pattern matching
type argument_pm is variant(pattern1:node_pattern1 ; pattern2:node_pattern2)

!results_matching is the type of the result from
!the pattern matching of a pattern
type results_matching is variant(without_var:bool ; with_var:singles)
```

```
!types for the log system
rec type games is variant(cons:facts ; empty:null) ! type of a game
&        facts is structure(fact:fact_bases ; next:games)

!type of the user moves
rec type moves is variant(cons:each_move ; empty:null)
& each_move is structure(who:string ; pos1:int ; pos2:int ; next:moves)
rec type perfbs is variant(cons:perfb ; empty:null)
&        perfb is structure(name:string ; node:games ; next:perfbs)
rec type permoves is variant(cons:permove ; empty:null)
&        permove is structure(name:string ; node:moves ; next:permoves)

!DECLARATION OF GLOBAL VARIABLES
let game := games(empty:nil)
let read_game := games(empty:nil)
let move := moves(empty:nil)
let read_move := moves(empty:nil)

!DECLARATION OF PROCEDURES
!AUXILIAR PROCEDURES
!PROCEDURES FOR THE LOG SYSTEM
let copy_state_game = proc(ff:fact_bases -> fact_bases)
begin
!local procedure copy_board
rec let copy_board = proc(bb,bb1:boards -> boards)
begin
 if bb is empty then bb1 := bb1
 else bb1 := boards(cons:node_boards(bb'cons(position),bb'cons(color),copy_board(bb'cons(next),bb1)))
 bb1
end

!main body of copy_state_game
let fres := fact_bases(ff(start),ff(move),ff(color),copy_board(ff(board),boards(empty:nil)))
 fres
end

rec let add_move = proc(who:string ; p1,p2:int ; move:moves -> moves)
begin
 if move is empty then move := moves(cons:each_move(who,p1,p2,move))
 else move'cons(next) := add_move(who,p1,p2,move'cons(next))
 move
end

rec let add_game = proc(fb:fact_bases ; game:games -> games)
begin
rec let create_board_game = proc(bb,bb1:boards -> boards)
begin
 if bb is empty then bb1 := bb1
 else bb1 := boards(cons:node_boards(bb'cons(position),bb'cons(color),
                                     create_board_game(bb'cons(next),bb1)))
 bb1
end
 if game is empty then game := games(cons:facts(fact_bases(fb(start),fb(move),fb(color),
                                                           create_board_game(fb(board),
                                                           boards(empty:nil))),
                                                           games(empty:nil)))
 else game'cons(next) := add_game(fb,game'cons(next))
 game
end
```

```
let store_game = proc(na:node_agenda ; fb:fact_bases)
begin
 if (na(rule_id) ~= "end1" and na(rule_id) ~= "end2") then
  game := add_game(fb,game)
 else
 begin
  writeString("Do you want to keep this last game?'n")
  writeString("If yes, then give the name of the game, otherwise write no'n")
  let answer := readString()
  if answer ~= "no" do
  begin
   if ~(games_env contains log_fb:perfbs and
        games_env contains log_move:permoves) do
   begin
    in games_env let log_fb := perfbs(empty:nil)
    in games_env let log_move := permoves(empty:nil)
   end
   use games_env with log_fb:perfbs ; log_move:permoves in
   begin
    log_fb := perfbs(cons:perfb(answer,game,log_fb))
    log_move := permoves(cons:permove(answer,move,log_move))
   end
  end
  game := games(empty:nil)
  move := moves(empty:nil)
 end
end

let log_system = proc(rule:node_agenda) !log_system is for the log system of games
begin
!local procedures find_move and find_game
rec let find_move = proc(log_move:permoves ; res:moves ; nn:string -> moves)
begin
 if log_move is empty then
  writeString("'nERROR: THAT GAME DOES NOT EXIST'n")
 else
  if log_move'cons(name) = nn then res := log_move'cons(node)
  else res := find_move(log_move'cons(next),res,nn)
 res
end

rec let find_game = proc(log_fb:perfbs ; res:games ; nn:string -> games)
begin
 if log_fb is empty then
  writeString("'nERROR: THAT GAME DOES NOT EXIST'n")
 else
  if log_fb'cons(name) = nn then res := log_fb'cons(node)
  else res := find_game(log_fb'cons(next),res,nn)
 res
end
!main body of log_system
 if (rule(rule_id)="start1" or rule(rule_id)="start2" or
    rule(rule_id)="start3") do
 begin
  read_move := moves(empty:nil)
  read_game := games(empty:nil)
  writeString("'nIf you want to replay a game write its name, ")
  writeString("otherwise write no:'n")
```

```
   let answer := readString()
   if answer ~="no" do
   begin
    if games_env contains log_move and games_env contains log_fb then
     use games_env with log_move:permoves ; log_fb:perfbs in
     begin
      read_move := find_move(log_move,read_move,answer)
      read_game := find_game(log_fb,read_game,answer)
     end
    else writeString("THERE ARE NOT YET GAMES KEPT TO BE REPLAYED")
   end
  end
end

let print_rules = proc(rb:rule_bases) !print_rules prints the rule base
begin
!local procedures
let writeList1 = proc(l:list1)
while l isnt empty do
begin
 writeString(l'cons(fact_name)) ; writeString(" ")
 writeString(l'cons(fact_value))
 writeString("'n")
 l := l'cons(next)
end
let writeList2 = proc(l:list2)
while l isnt empty do
begin
 if l'cons(position) is pos_var then
  {writeString(l'cons(position)'pos_var) ;  writeString(" ")}
 else writeInt(l'cons(position)'pos_const)
 writeString(l'cons(color))
 writeString("'n")
 l := l'cons(next)
end
let writeTest = proc(t:test) ; if t is test_id do writeString(t'test_id)
!main body of procedure print_rules
while rb isnt empty do
begin
 let rule := rb'cons
 writeString(rule(rule_id)) ;    writeString("'n")
 writeList1(rule(pattern1))
 writeList2(rule(pattern2))
 writeTest(rule(pattern3))  ;    writeString("'n")
 writeString(rule(right_side)) ; writeString("'n") ; writeString("'n")
 rb := rule(next)
end
end

! procedures for print_agenda
rec let print_singles = proc(s:list3)
 if s isnt empty do
 begin
  writeInt(s'cons(position))
  print_singles(s'cons(next))
 end

rec let print_tuples = proc(tu:list4)
 if tu isnt empty do
```

```
 begin
  writeInt(tu'cons(position1))
  writeInt(tu'cons(position2))
  writeString("'n")
  print_tuples(tu'cons(next))
 end

rec let print_triples = proc(tr:list5)
 if tr isnt empty do
 begin
  writeInt(tr'cons(position1))
  writeInt(tr'cons(position2))
  writeInt(tr'cons(position3))
  writeString("'n")
  print_triples(tr'cons(next))
 end

let print_elem = proc(elem:elem_type)
begin
 if elem is elem0 do writeBool(elem'elem0) ; writeString("'n")
 if elem is elem1 do
 begin
  writeString(elem'elem1(var_pos))
  print_singles(elem'elem1(pos_list)) ; writeString("'n")
 end
 if elem is elem2 do
 begin
  writeString(elem'elem2(var_pos1))
  writeString(elem'elem2(var_pos2))
  print_tuples(elem'elem2(tuples_list)) ; writeString("'n")
 end
 if elem is elem3 do
 begin
  writeString(elem'elem3(var_pos1))
  writeString(elem'elem3(var_pos2))
  writeString(elem'elem3(var_pos3))
  print_triples(elem'elem3(triples_list)) ; writeString("'n")
 end
end

!procedure print_agenda
rec let print_agenda = proc(ag:agendas)
begin
 if ag isnt empty do
 begin
  writeString(ag'cons(rule_id)) ; writeString("'n")
  print_elem(ag'cons(agenda_elem))
  print_agenda(ag'cons(next))
 end
end

!PROCEDURES RELATED TO THE RESOLUTION OF THE PROBLEM
!create_rules creates the rule base
let create_rules = proc(rule_base:rule_bases -> rule_bases)
begin
rule_base := rule_bases(cons:
node_rule("start1",
          list1(cons:node_pattern1("start","yes",
               list1(cons:node_pattern1("color",",
```

```
                list1(empty:nil))))),
           list2(empty:nil),
           test(empty:nil),
           "action_start1",
           rule_bases(cons:
node_rule("start2",
           list1(cons:node_pattern1("start","yes",
                list1(cons:node_pattern1("color","white",
                list1(empty:nil))))),
           list2(empty:nil),
           test(empty:nil),
           "action_start2",
           rule_bases(cons:
node_rule("start3",
           list1(cons:node_pattern1("start","yes",
                list1(cons:node_pattern1("color","black",
                list1(empty:nil))))),
           list2(empty:nil),
           test(empty:nil),
           "action_start3",
           rule_bases(cons:
node_rule("end1",
           list1(empty:nil),
           list2(cons:node_pattern2(varcons(pos_const:0),"white",
                list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
                list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"k"),"white",
                list2(empty:nil)))))))))),
           test(test_id:"middle"),
           "action_end1",
           rule_bases(cons:
node_rule("end2",
           list1(empty:nil),
           list2(cons:node_pattern2(varcons(pos_const:0),"black",
                list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
                list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
                list2(cons:node_pattern2(varcons(pos_var:"k"),"black",
                list2(empty:nil)))))))))),
           test(test_id:"middle"),
           "action_end2",
           rule_bases(cons:
node_rule("user",
           list1(cons:node_pattern1("move","user",
                list1(empty:nil))),
           list2(empty:nil),
           test(empty:nil),
           "action_user",
           rule_bases(cons:
node_rule("move_to_putahi1",
           list1(cons:node_pattern1("move","machine",
                list1(cons:node_pattern1("color","white",
                list1(empty:nil))))),
           list2(cons:node_pattern2(varcons(pos_const:0),"empty",
                list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
                list2(empty:nil)))))))),
           test(test_id:"adjacent"),
           "action_move_to_putahi1",
           rule_bases(cons:
```

```
node_rule("move_to_putahi2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))))),
          list2(cons:node_pattern2(varcons(pos_const:0),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
               list2(empty:nil)))))))),
          test(test_id:"adjacent"),
          "action_move_to_putahi2",
          rule_bases(cons:
node_rule("move_from_putahi1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))))),
          list2(cons:node_pattern2(varcons(pos_const:0),"white",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
               list2(empty:nil)))))),
          test(empty:nil),
          "action_move_from_putahi1",
          rule_bases(cons:
node_rule("move_from_putahi2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))))),
          list2(cons:node_pattern2(varcons(pos_const:0),"black",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
               list2(empty:nil)))))),
          test(empty:nil),
          "action_move_from_putahi2",
          rule_bases(cons:
node_rule("move_adjacent1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))))),
          list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"empty",
               list2(empty:nil)))))),
          test(test_id:"adjacent"),
          "action_move_adjacent1",
          rule_bases(cons:
node_rule("move_adjacent2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))))),
          list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"empty",
               list2(empty:nil)))))),
          test(test_id:"adjacent"),
          "action_move_adjacent2",
          rule_bases(empty:nil))))))))))))))))))))))))))))
rule_base
end

!FUNCTIONS OF THE LEFT HAND SIDE OF THE RULES
let adjacent = proc(i,j:int -> bool)
begin
 let result := false
 if i=0 or j=0 then result := false
```

```
 else
  case i of
          1: if j=8 or j=2 then result := true
             else result := false
          8: if j=1 or j=7 then result := true
             else result := false
     default: if abs(i-j)=1 then result := true
              else result := false
 result
end

let middle = proc(i,j,k:int -> bool)
begin
 let result := false
 if i=0 or j=0 or k=0 then result := false
 else
  case i of
          1: if (j=8 and k=2) or (j=2 and k=8) then result := true
             else result := false
          8: if (j=1 and k=7) or (j=7 and k=1) then result := true
             else result := false
     default: if (j=(i+1) and k=(i-1)) or (j=(i-1) and k=(i+1))
              then result := true
              else result := false
 result
end

let initialise_game = proc( -> fact_bases) !initialisation of the game
begin
 let fact_base := fact_bases("yes","","",boards(empty:nil))
 fact_base
end

rec let print_board = proc(board:boards) !printing of the board
 if board isnt empty do
 begin
  writeString("BOARD ")
  writeInt(board'cons(position))
  writeString(board'cons(color)) ; writeString("'n")
  print_board(board'cons(next))
 end

let print_state_of_game = proc(fb:fact_bases) !printing of the state of the game
begin
 writeString("The state of the game is:'n")
 print_board(fb(board))                    !print board after each turn
 writeString("The next player is: ")
 writeString(fb(move)) ; writeString("'n")  !print who's turn it is next
 writeString("Condition of start: ")
 writeString(fb(start)) ; writeString("'n") !print the condition of start
 writeString("The colour of the machine is: ")
 writeString(fb(color)) ; writeString("'n")  !print colour of the machine
end

rec let print_game = proc(game:games) !printing of the state of the game being logged
if game isnt empty then
begin
 print_state_of_game(game'cons(fact))
 print_game(game'cons(next))
```

```
 end
 else writeString("THE GAME BEING LOGGED IS EMPTY")

 !SET OF PROCEDURES FOR THE RIGHT HAND SIDE OF THE RULES
 let create_initial_board = proc( -> boards)
 begin
  let board := boards(cons:
               node_boards(0,"empty",boards(cons:
               node_boards(1,"white",boards(cons:
               node_boards(2,"white",boards(cons:
               node_boards(3,"white",boards(cons:
               node_boards(4,"white",boards(cons:
               node_boards(5,"black",boards(cons:
               node_boards(6,"black",boards(cons:
               node_boards(7,"black",boards(cons:
               node_boards(8,"black",boards(empty:nil))))))))))))))))))
  board
 end

 let ask_user_continue = proc( -> bool)
 begin
  let result := false
  writeString("Do you want to continue?. Answer yes or no:'n")
  if readString() = "yes" then result := true
  else result := false
  result
 end

 let reinitialise_fact_base = proc(fb:fact_bases -> fact_bases)
 begin
  fb(start) := "yes"
  fb(move) := ""
  fb(board) := boards(empty:nil)
  fb
 end

 let delete_fact_base = proc(fb:fact_bases -> fact_bases)
 begin
  fb(start) := ""
  fb(move) := ""
  fb(color) := ""
  fb(board) := boards(empty:nil)
  fb
 end

 let machineWins = proc()
 begin
 writeString("The machine wins.'n")
 end

 let userWins = proc()
 begin
 writeString("The user wins.'n")
 end

 let modify_board = proc(b:boards ; p1,p2:int ; c1,c2:string -> boards)
 begin
 !local procedure
 rec let modify_position = proc(b:boards ; p:int ; c:string -> boards)
```

```
begin
 if b'cons(position) = p then b'cons(color) := c
 else b'cons(next) := modify_position(b'cons(next),p,c)
 b
end
!main body of procedure modify_board
 b := modify_position(b,p1,c1)
 b := modify_position(b,p2,c2)
 b
end

let action_start1 = proc(fb:fact_bases -> fact_bases)
begin
!local procedures
let ask_user_start = proc( -> bool)
begin
 let result := false
 writeString("Do you want to be the first to start the game?'n")
 writeString("Answer yes or no: ")
 if readString() = "yes" then result := true
 else result := false
 result
end
!main body of procedure action_start1
if read_game is empty then
begin
 fb(start) := ""
 fb(board) := create_initial_board()
 if ask_user_start() then {fb(color) := "white" ; fb(move) := "user"}
 else {fb(color) := "black" ; fb(move) := "machine"}
end
else fb := copy_state_game(read_game'cons(fact))
fb
end

let action_start2 = proc(fb:fact_bases -> fact_bases)
begin
 if read_game is empty then
 begin
  fb(start) := ""
  fb(move) := "machine"
  fb(color) := "black"
  fb(board) := create_initial_board()
 end
 else fb := copy_state_game(read_game'cons(fact))
 fb
end

let action_start3 = proc(fb:fact_bases -> fact_bases)
begin
 if read_game is empty then
 begin
  fb(start) := ""
  fb(move) := "user"
  fb(color) := "white"
  fb(board) := create_initial_board()
 end
 else fb := copy_state_game(read_game'cons(fact))
 fb
```

```
 end

 let action_end1 = proc(fb:fact_bases -> fact_bases)
 begin
 if fb(color) = "white" then machineWins()
 else userWins()
 if ask_user_continue() then fb := reinitialise_fact_base(fb)
 else fb := delete_fact_base(fb)
 fb
 end

 let action_end2 = proc(fb:fact_bases -> fact_bases)
 begin
 if fb(color) = "black" then machineWins()
 else userWins()
 if ask_user_continue() then fb := reinitialise_fact_base(fb)
 else fb := delete_fact_base(fb)
 fb
 end

 let action_user = proc(fb:fact_bases -> fact_bases)
 begin
  let pos1 := 10 ; let pos2 :=10
  if read_move is empty then
  begin
   writeString("'n It is your turn to move. Indicate initial position: ")
   pos1 := readInt()
   writeString("'n Indicate final position: ")
   pos2 := readInt()
  end
  else
  begin
   if read_move'cons(who)="machine" do read_move := read_move'cons(next)
   pos1 := read_move'cons(pos1)
   pos2 := read_move'cons(pos2)
   writeString("NEXT MOVE READ FROM THE LOG_MOVE IS'n")
   writeString(read_move'cons(who))
   writeString("'n")
   writeInt(pos1)
   writeString("'n")
   writeInt(pos2)
   writeString("'n")
   writeString("END OF NEXT MOVE READ FROM THE LOG_MOVE IS'n")
   read_move := read_move'cons(next)
  end
  let c := ""
  case fb(color) of
      "white": c := "black"
      "black": c := "white"
      default: {}
  move := add_move(fb(move),pos1,pos2,move)
  fb(board) := modify_board(fb(board),pos1,pos2,"empty",c)
  fb(move) := "machine"
  fb
 end

 let action_move_to_putahi1 = proc(fb:fact_bases ; i:int -> fact_bases)
 begin
  move := add_move(fb(move),i,0,move)
```

```
 fb(board) := modify_board(fb(board),0,i,"white","empty")
 fb(move) := "user"
 fb
end

let action_move_to_putahi2 = proc(fb:fact_bases ; i:int -> fact_bases)
begin
 move := add_move(fb(move),i,0,move)
 fb(board) := modify_board(fb(board),0,i,"black","empty")
 fb(move) := "user"
 fb
end

let action_move_from_putahi1 = proc(fb:fact_bases ; i:int -> fact_bases)
begin
 move := add_move(fb(move),0,i,move)
 fb(board) := modify_board(fb(board),0,i,"empty","white")
 fb(move) := "user"
 fb
end

let action_move_from_putahi2 = proc(fb:fact_bases ; i:int -> fact_bases)
begin
 move := add_move(fb(move),0,i,move)
 fb(board) := modify_board(fb(board),0,i,"empty","black")
 fb(move) := "user"
 fb
end

let action_move_adjacent1 = proc(fb:fact_bases ; i,j:int -> fact_bases)
begin
 move := add_move(fb(move),i,j,move)
 fb(board) := modify_board(fb(board),i,j,"empty","white")
 fb(move) := "user"
 fb
end

let action_move_adjacent2 = proc(fb:fact_bases ; i,j:int -> fact_bases)
begin
 move := add_move(fb(move),i,j,move)
 fb(board) := modify_board(fb(board),i,j,"empty","black")
 fb(move) := "user"
 fb
end

!ENGINE
!PATTERN MATCHING OF EACH PATTERN OF A RULE
let pattern_matching = proc(arg:argument_pm ; fb:fact_bases -> results_matching)
begin
!local procedures
let put_facts_in_wm = proc(fn:string ; wm:wms ; fb:fact_bases -> wms)
begin !local procedure
rec let create_board_wm = proc(bb,bb1:boards -> boards)
begin
 if bb is empty then bb1 := bb1
 else bb1 := boards(cons:node_boards(bb'cons(position),bb'cons(color),
                                     create_board_wm(bb'cons(next),bb1)))
 bb1
end
```

```
 case fn of
     "start": wm := wms(start:fb(start))
     "color": wm := wms(color:fb(color))
     "move" : wm := wms(move:fb(move))
     "board": wm := wms(board:create_board_wm(fb(board),boards(empty:nil)))
     default: {}
 wm
end

let print_wm = proc(wm:wms)
begin
 if wm is start do {writeString(wm'start) ; writeString("'n")}
 if wm is color do {writeString(wm'color) ; writeString("'n")}
 if wm is move do {writeString(wm'move) ; writeString("'n")}
 if wm is board do {print_board(wm'board) ; writeString("was board'n")}
end

let match_constants = proc(wm:wms ; arg:argument_pm -> wms)
begin
!local procedures
let compare_pos = proc(wm:wms ; pos:int -> wms)
begin
 while wm'board'cons(position) ~= pos do
 begin
  wm := wms(board:wm'board'cons(next))
 end
 wm'board'cons(next) := boards(empty:nil)
 wm
end
let compare_color = proc(wm:wms ; color:string -> wms)
begin
 let aux := boards(empty:nil)
 while wm'board isnt empty do
 begin
  if wm'board'cons(color) = color do
   aux := boards(cons:node_boards(wm'board'cons(position),wm'board'cons(color),aux))
  wm := wms(board:wm'board'cons(next))
 end
 if aux is empty then wm := wms(empty:nil)
 else wm := wms(board:aux)
 wm
end
!main body of match_constants
 if wm is start do
  if arg'pattern1(fact_value) ~= wm'start do wm := wms(empty:nil)
 if wm is color do
  if arg'pattern1(fact_value) ~= wm'color do wm := wms(empty:nil)
 if wm is move do
  if arg'pattern1(fact_value) ~= wm'move do wm := wms(empty:nil)
 if wm is board do
 begin
  if arg'pattern2(position) is pos_const do
   wm := compare_pos(wm,arg'pattern2(position)'pos_const)
  wm := compare_color(wm,arg'pattern2(color))
 end
 wm
end
let match_variables = proc(wm:wms ; bb:node_pattern2 -> singles)
begin
```

```
 let list_of_values := singles("",list3(empty:nil))
 if bb(position) is pos_var do
 begin
  list_of_values(var_pos) := bb(position)'pos_var
  while wm'board isnt empty do
  begin
   list_of_values(pos_list) :=
    list3(cons:node_pos(wm'board'cons(position),list_of_values(pos_list)))
   wm := wms(board:wm'board'cons(next))
  end
 end
 list_of_values
end
!main body of the procedure pattern_matching
 let wm := wms(empty:nil)
 let result := results_matching(without_var:true)
 let list_of_values := singles("",list3(empty:nil))
 if arg is pattern1 then
 begin
  let fn := arg'pattern1(fact_name)
  wm := put_facts_in_wm(fn,wm,fb)
  wm := match_constants(wm,arg)
  if wm is empty do result := results_matching(without_var:false)
 end
 else
 begin
  wm := put_facts_in_wm("board",wm,fb)
  wm := match_constants(wm,arg)
  if wm is empty then result := results_matching(without_var:false)
  else
  begin
   list_of_values := match_variables(wm,arg'pattern2)
   if list_of_values(var_pos) ~= "" do
    if list_of_values(pos_list) is empty then
     result := results_matching(without_var:false)
    else result := results_matching(with_var:list_of_values)
  end
 end
 result
end

!DETECTION OF RULES THAT ACCOMPLISH THE FACT BASE
let detection = proc(rb:rule_bases ; fb:fact_bases -> agendas)
begin
!local procedures
let form_tuples = proc(s1,s2:singles -> tuples) !Form elements of the agenda
begin !local procedures
rec let sub_tuples = proc(x:int ; l2:list3 ; l3:list4 -> list4)
begin
 if l2 is empty then l3 := l3
 else l3 := list4(cons:node_tuples(x,l2'cons(position),sub_tuples(x,l2'cons(next),l3)))
 l3
end
rec let do_tuples = proc(l1,l2:list3 ; l3:list4 -> list4)
begin
 if l1 is empty then l3 := l3
 else l3 := sub_tuples(l1'cons(position),l2,do_tuples(l1'cons(next),l2,l3))
 l3
end
```

```
!main body of form_tuples
 let t := tuples(s1(var_pos),s2(var_pos),
                 do_tuples(s1(pos_list),s2(pos_list),list4(empty:nil)))
 t
end

let form_triples = proc(s1,s2,s3:singles -> triples)
begin !local procedures
rec let sub_trip2 = proc(x,y:int ; l3:list3 ; l4 :list5 -> list5)
begin
 if l3 is empty then l4 := l4
 else l4 := list5(cons:node_triples(x,y,l3'cons(position),sub_trip2(x,y,l3'cons(next),l4)))
 l4
end
rec let sub_trip1 = proc(x:int ; l2,l3:list3 ; l4:list5 -> list5)
begin
 if l2 is empty then l4:= l4
 else l4 := sub_trip2(x,l2'cons(position),l3,sub_trip1(x,l2'cons(next),l3,l4))
 l4
end
rec let do_triples = proc(l1,l2,l3:list3 ; l4:list5 -> list5)
begin
 if l1 is empty then l4 := l4
 else
  l4 := sub_trip1(l1'cons(position),l2,l3,do_triples(l1'cons(next),l2,l3,l4))
 l4
end
!main body of form_triples
 let t := triples(s1(var_pos),s2(var_pos),s3(var_pos),
                  do_triples(s1(pos_list),s2(pos_list),s3(pos_list),list5(empty:nil)))
 t
end

!functions for the lhs of the rules applied to the list of possibilities
rec let adjacent_of_list = proc(l1,l2:list4 -> list4)
begin
 if l1 is empty then l2 := l2
 else
 begin
  let i := l1'cons(position1)
  let j := l1'cons(position2)
  if adjacent(i,j) do l2 := list4(cons:node_tuples(i,j,l2))
  l2 := adjacent_of_list(l1'cons(next),l2)
 end
 l2
end

rec let middle_of_list = proc(l1,l2:list5 -> list5)
begin
 if l1 is empty then l2 := l2
 else
 begin
  let i := l1'cons(position1)
  let j := l1'cons(position2)
  let k := l1'cons(position3)
  if middle(i,j,k) do l2 := list5(cons:node_triples(i,j,k,l2))
  l2 := middle_of_list(l1'cons(next),l2)
 end
 l2
```

```
end

!append procedures
rec let append_singles = proc(pr:patterns_results ; r:singles -> patterns_results)
begin
 if pr is empty then pr := patterns_results(cons:node_results(r,pr))
 else pr'cons(next) := append_singles(pr'cons(next),r)
 pr
end

rec let append_agenda = proc(ag,final_list:agendas -> agendas)
begin
 if ag is empty then ag := final_list
 else ag'cons(next) := append_agenda(ag'cons(next),final_list)
 ag
end
!main body of detection
 let agenda := agendas(empty:nil)
 while rb isnt empty do
 begin
 let rule := node_rule(rb'cons(rule_id),
                       rb'cons(pattern1),
                       rb'cons(pattern2),
                       rb'cons(pattern3),
                       rb'cons(right_side),
                       rule_bases(empty:nil))
 let abandon_rule := false
 let list_of_results := patterns_results(empty:nil)
 let final_list := agendas(empty:nil)
 let list_pattern1 := rule(pattern1)
 let list_pattern2 := rule(pattern2)
 while list_pattern1 isnt empty and ~abandon_rule do
 begin
  let result := pattern_matching(argument_pm(pattern1:list_pattern1'cons),fb)
  if result'without_var=false do abandon_rule := true
  list_pattern1 := list_pattern1'cons(next)
 end
 while list_pattern2 isnt empty and ~abandon_rule do
 begin
  if fb(board) is empty then abandon_rule := true
  else
  begin
   let result := pattern_matching(argument_pm(pattern2:list_pattern2'cons),fb)
   if result is with_var then
    list_of_results := append_singles(list_of_results,result'with_var)
   else
    if result'without_var=false do abandon_rule := true
   list_pattern2 := list_pattern2'cons(next)
  end
 end
  if ~abandon_rule do
  begin
   if list_of_results is empty then
   begin
    final_list := agendas(cons:node_agenda(rule(rule_id),elem_type(elem0:true),agendas(empty:nil)))
    agenda := append_agenda(agenda,final_list)
   end
   else
   begin
```

```
    let l := list_of_results'cons
    if l(next) is empty then
    begin
     final_list := agendas(cons:node_agenda(rule(rule_id),elem_type(elem1:l(results)),
                                        agendas(empty:nil)))
     agenda := append_agenda(agenda,final_list)
    end
    else
    begin
     let ll := l(next)'cons
     if ll(next) is empty then
     begin
      final_list:=agendas(cons:node_agenda(rule(rule_id),elem_type(elem2:
                             form_tuples(l(results),ll(results))),agendas(empty:nil)))
      final_list'cons(agenda_elem)'elem2(tuples_list) :=
       adjacent_of_list(final_list'cons(agenda_elem)'elem2(tuples_list),list4(empty:nil))
      if final_list'cons(agenda_elem)'elem2(tuples_list) isnt empty do
       agenda := append_agenda(agenda,final_list)
     end
     else
     begin
      let lll := ll(next)'cons
      final_list := agendas(cons:node_agenda(rule(rule_id),
                    elem_type(elem3:form_triples(l(results),ll(results),lll(results))),
                                        agendas(empty:nil)))
      final_list'cons(agenda_elem)'elem3(triples_list) :=
       middle_of_list(final_list'cons(agenda_elem)'elem3(triples_list),list5(empty:nil))
      if final_list'cons(agenda_elem)'elem3(triples_list) isnt empty do
       agenda := append_agenda(agenda,final_list)
     end
    end
   end
  end
  rb := rb'cons(next)
 end
 agenda
end

!CHOICE OF THE RULE TO BE FIRED BETWEEN ALL RULES DETECTED AS GOOD
let choice = proc(ag:agendas -> node_agenda)
begin
 let na := node_agenda(ag'cons(rule_id),ag'cons(agenda_elem),agendas(empty:nil))
 na
end

!EXECUTION OF THE RULE CHOSEN
let deduction = proc(na:node_agenda ; fb:fact_bases -> fact_bases)
begin
 let elem := na(agenda_elem)
 if elem is elem0 do
  case na(rule_id) of
      "start1": fb := action_start1(fb)
      "start2": fb := action_start2(fb)
      "start3": fb := action_start3(fb)
        "user": fb := action_user(fb)
       default:{}
 if elem is elem1 do
 begin
  let i := elem'elem1(pos_list)'cons(position)
```

```
  case na(rule_id) of
      "move_from_putahi1": fb := action_move_from_putahi1(fb,i)
      "move_from_putahi2": fb := action_move_from_putahi2(fb,i)
                 default:{}
 end
 if elem is elem2 do
 begin                                !it assumes that the order is (i,j)
  let i := elem'elem2(tuples_list)'cons(position1)
  let j := elem'elem2(tuples_list)'cons(position2)
  if elem'elem2(var_pos1) = "j" and elem'elem2(var_pos2) = "i" do
  { let p := i ; i := j ; j := p}   ! it happens that the order is (j,i)
  case na(rule_id) of
      "move_to_putahi1": fb := action_move_to_putahi1(fb,i)
      "move_to_putahi2": fb := action_move_to_putahi2(fb,i)
       "move_adjacent1": fb := action_move_adjacent1(fb,i,j)
       "move_adjacent2": fb := action_move_adjacent2(fb,i,j)
              default:{}
 end
 if elem is elem3 do
  case na(rule_id) of
      "end1": fb := action_end1(fb)
      "end2": fb := action_end2(fb)
     default:{}
 store_game(na,fb)
 print_state_of_game(fb)
 fb
end

let engine = proc(rule_base:rule_bases ; fact_base:fact_bases)
begin
 let finish := false
 while ~finish do
 begin
  let agenda := detection(rule_base,fact_base)
  if agenda is empty then finish := true
  else
  begin
   let chosen_rule := choice(agenda)
   log_system(chosen_rule)
   fact_base := deduction(chosen_rule,fact_base)
  end
 end
end

!MAIN BODY OF PROGRAM
 let rule_base := rule_bases(empty:nil)
 rule_base := create_rules(rule_base)   ! print_rules(rule_base)
 let fact_base := initialise_game()
 engine(rule_base,fact_base)
end
```

E.2 Prototype Two

```
!declaration of persistent environments
let ps = PS()
use ps with IO,Arithmetical,Time,User:env in
use Arithmetical with abs:proc(int -> int) in
use IO with writeString: proc(string) ; writeInt:proc(int) ;
            writeBool:proc(bool) ;
            readString:proc( -> string) ; readInt:proc( -> int) in
use Time with time:proc( -> int) in
use User with torere_env:env in
use torere_env with log_env:env in
use log_env with games2_env:env in
begin

!TYPES OF THE FACT BASE
!color contains the colour the program's pereperes: empty, white or black.
!move indicates whose turn is to move: program or user.
!start indicates if it is the begining of a game or not: yes, no.
!boards is the type of the state of the board in the game.

rec type boards is variant(cons:node_boards ; empty:null)
&      node_boards is structure(position:int ; color:string ; next:boards)
!fact_bases is the type of the fact base
type fact_bases is structure(start:string ;
                             move:string ;
                             color:string ;
                             board:boards)

!TYPES OF RULE BASE
!type of pattern1
rec type list1 is variant(cons:node_pattern1 ; empty:null)
&        node_pattern1 is structure(fact_name:string ;
                                    fact_value:string ;
                                    next:list1)
!types for pattern2
type varcons is variant(pos_var:string ; pos_const:int)
rec type list2 is variant(cons:node_pattern2 ; empty:null)
&        node_pattern2 is structure(position:varcons ;
                                    color:string ;
                                    next:list2)
!type of pattern3
type test is variant(test_id:string ; empty:null)
!rule_bases is the type of the rule base
rec type rule_bases is variant(cons:node_rule ; empty:null)
&        node_rule is structure(rule_id:string ;
                                pattern1:list1 ;
                                pattern2:list2 ;
                                pattern3:test ;
                                right_side:string ;
                                next:rule_bases)

!TYPES OF WORKING MEMORY
!wms is the type of the working memory
type wms is variant(board:boards ;
                    color:string ;
                    move:string ;
```

```
                    start:string ;
                    empty:null)

!TYPES OF THE AGENDA
!types for elem1
rec type list3 is variant(cons:node_pos ; empty:null)
&       node_pos is structure(position:int ; next:list3)
type singles is structure(var_pos:string ; pos_list:list3)
!types for elem2
rec type list4 is variant(cons:node_tuples ; empty:null)
&       node_tuples is structure(position1:int ; position2:int ; next:list4)
type tuples is structure(var_pos1:string ; var_pos2:string ; tuples_list:list4)
!types for elem3
rec type list5 is variant(cons:node_triples ; empty:null)
&       node_triples is structure(position1:int ;
                                  position2:int ;
                                  position3:int ;
                                  next:list5)
type triples is structure(var_pos1:string ;
                          var_pos2:string ;
                          var_pos3:string ;
                          triples_list:list5)
!elem_type is the type of the elements of the agenda
type elem_type is variant(elem0:bool ; elem1:singles ; elem2:tuples ; elem3:triples)
!agendas is the type of the input elements for the agenda
rec type agendas is variant(cons:node_agenda ; empty:null)
&       node_agenda is structure(rule_id:string ;
                                 agenda_elem:elem_type ;
                                 next:agendas)

!OTHER TYPES FOR THE PROGRAM
!patterns_results is the type of the element containing
!all the variable results of a rule
rec type patterns_results is variant(cons:node_results ; empty:null)
& node_results is structure(results:singles ; next:patterns_results)

!argument_pm is the type of the argument to the pattern matching
type argument_pm is variant(pattern1:node_pattern1 ; pattern2:node_pattern2)
!results_matching is the type of the result from
!the pattern matching of a pattern
type results_matching is variant(without_var:bool ; with_var:singles)

!types for the log system
rec type games is variant(cons:facts ; empty:null) ! type of a game
&       facts is structure(fact:fact_bases ; next:games)

!type of the user moves
rec type moves is variant(cons:each_move ; empty:null)
& each_move is structure(who:string ; pos1:int ; pos2:int ; next:moves)
rec type perfbs is variant(cons:perfb ; empty:null)
&       perfb is structure(name:string ; node:games ; next:perfbs)
rec type permoves is variant(cons:permove ; empty:null)
&       permove is structure(name:string ; node:moves ; next:permoves)

!DECLARATION OF GLOBAL VARIABLES

let game := games(empty:nil)
let read_game := games(empty:nil)
let move := moves(empty:nil)
```

```
let read_move := moves(empty:nil)

!DECLARATION OF PROCEDURES
!AUXILIAR PROCEDURES
!PROCEDURES FOR THE LOG SYSTEM
let copy_state_game = proc(ff:fact_bases -> fact_bases)
begin
!local procedure copy_board
rec let copy_board = proc(bb,bb1:boards -> boards)
begin
 if bb is empty then bb1 := bb1
 else bb1 := boards(cons:node_boards(bb'cons(position),bb'cons(color),copy_board(bb'cons(next),bb1)))
 bb1
end

!main body of copy_state_game
let fres := fact_bases(ff(start),ff(move),ff(color),copy_board(ff(board),boards(empty:nil)))
 fres
end

rec let add_move = proc(who:string ; p1,p2:int ; move:moves -> moves)
begin
 if move is empty then move := moves(cons:each_move(who,p1,p2,move))
 else move'cons(next) := add_move(who,p1,p2,move'cons(next))
 move
end

rec let add_game = proc(fb:fact_bases ; game:games -> games)
begin
rec let create_board_game = proc(bb,bb1:boards -> boards)
begin
 if bb is empty then bb1 := bb1
 else bb1 := boards(cons:node_boards(bb'cons(position),bb'cons(color),
                                     create_board_game(bb'cons(next),bb1)))
 bb1
end
 if game is empty then game := games(cons:facts(fact_bases(fb(start),fb(move),fb(color),
                                  create_board_game(fb(board),boards(empty:nil))),games(empty:nil)))
 else game'cons(next) := add_game(fb,game'cons(next))
 game
end

let store_game = proc(na:node_agenda ; fb:fact_bases)
begin
 if (na(rule_id) ~= "end1" and na(rule_id) ~= "end2") then
  game := add_game(fb,game)
 else
 begin
  writeString("Do you want to keep this last game?'n")
  writeString("If yes, then give the name of the game, otherwise write no'n")
  let answer := readString()
  if answer ~= "no" do
  begin
   if ~(games2_env contains log_fb:perfbs and
        games2_env contains log_move:permoves) do
   begin
    in games2_env let log_fb := perfbs(empty:nil)
    in games2_env let log_move := permoves(empty:nil)
   end
```

```
    use games2_env with log_fb:perfbs ; log_move:permoves in
    begin
     log_fb := perfbs(cons:perfb(answer,game,log_fb))
     log_move := permoves(cons:permove(answer,move,log_move))
    end
   end
   game := games(empty:nil)
   move := moves(empty:nil)
  end
 end

 !log_system is for the log system of games
 let log_system = proc(rule:node_agenda)
 begin
 !local procedures find_move and find_game
 rec let find_move = proc(log_move:permoves ; res:moves ; nn:string -> moves)
 begin
  if log_move is empty then
   writeString("'nERROR: THAT GAME DOES NOT EXIST'n")
  else
   if log_move'cons(name) = nn then res := log_move'cons(node)
   else res := find_move(log_move'cons(next),res,nn)
  res
 end

 rec let find_game = proc(log_fb:perfbs ; res:games ; nn:string -> games)
 begin
  if log_fb is empty then
   writeString("'nERROR: THAT GAME DOES NOT EXIST'n")
  else
   if log_fb'cons(name) = nn then res := log_fb'cons(node)
   else res := find_game(log_fb'cons(next),res,nn)
  res
 end

 !main body of log_system
  if (rule(rule_id)="start1" or rule(rule_id)="start2" or rule(rule_id)="start3") do
  begin
   read_move := moves(empty:nil)
   read_game := games(empty:nil)
   writeString("'nIf you want to replay a game write its name, ")
   writeString("otherwise write no:'n")
   let answer := readString()
   if answer ~="no" do
   begin
    if games2_env contains log_move and games2_env contains log_fb
    then use games2_env with log_move:permoves ; log_fb:perfbs in
     begin
      read_move := find_move(log_move,read_move,answer)
      read_game := find_game(log_fb,read_game,answer)
     end
    else writeString("THERE ARE NOT YET GAMES KEPT TO BE REPLAYED")
   end
  end
 end

 !PRINTING PROCEDURES
 !print_rules prints the rule base
 let print_rules = proc(rb:rule_bases)
```

```
begin
!local procedures
let writeList1 = proc(l:list1)
while l isnt empty do
begin
 writeString(l'cons(fact_name)) ; writeString(" ")
 writeString(l'cons(fact_value))
 writeString("'n")
 l := l'cons(next)
end

let writeList2 = proc(l:list2)
while l isnt empty do
begin
 if l'cons(position) is pos_var then
  {writeString(l'cons(position)'pos_var) ;  writeString(" ")}
 else writeInt(l'cons(position)'pos_const)
 writeString(l'cons(color))
 writeString("'n")
 l := l'cons(next)
end

let writeTest = proc(t:test)
 if t is test_id do writeString(t'test_id)

!main body of procedure print_rules
while rb isnt empty do
begin
 let rule := rb'cons
 writeString(rule(rule_id)) ;    writeString("'n")
 writeList1(rule(pattern1))
 writeList2(rule(pattern2))
 writeTest(rule(pattern3))  ;    writeString("'n")
 writeString(rule(right_side)) ; writeString("'n") ; writeString("'n")
 rb := rule(next)
end
end

! procedures for print_agenda
rec let print_singles = proc(s:list3)
 if s isnt empty do
 begin
  writeInt(s'cons(position))
  print_singles(s'cons(next))
 end

rec let print_tuples = proc(tu:list4)
 if tu isnt empty do
 begin
  writeInt(tu'cons(position1))
  writeInt(tu'cons(position2))
  writeString("'n")
  print_tuples(tu'cons(next))
 end

rec let print_triples = proc(tr:list5)
 if tr isnt empty do
 begin
  writeInt(tr'cons(position1))
```

```
  writeInt(tr'cons(position2))
  writeInt(tr'cons(position3))
  writeString("'n")
  print_triples(tr'cons(next))
 end

let print_elem = proc(elem:elem_type)
begin
 if elem is elem0 do writeBool(elem'elem0) ; writeString("'n")
 if elem is elem1 do
 begin
  writeString(elem'elem1(var_pos))
  print_singles(elem'elem1(pos_list)) ; writeString("'n")
 end
 if elem is elem2 do
 begin
  writeString(elem'elem2(var_pos1))
  writeString(elem'elem2(var_pos2))
  print_tuples(elem'elem2(tuples_list)) ; writeString("'n")
 end
 if elem is elem3 do
 begin
  writeString(elem'elem3(var_pos1))
  writeString(elem'elem3(var_pos2))
  writeString(elem'elem3(var_pos3))
  print_triples(elem'elem3(triples_list)) ; writeString("'n")
 end
end

!procedure print_agenda
rec let print_agenda = proc(ag:agendas)
begin
 if ag isnt empty do
 begin
  writeString(ag'cons(rule_id)) ; writeString("'n")
  print_elem(ag'cons(agenda_elem))
  print_agenda(ag'cons(next))
 end
end

!PROCEDURES RELATED TO THE RESOLUTION OF THE PROBLEM
!create_rules creates the rule base
let create_rules = proc(rule_base:rule_bases -> rule_bases)
begin
rule_base := rule_bases(cons:
node_rule("start1",
          list1(cons:node_pattern1("start","yes",
               list1(cons:node_pattern1("color","",
               list1(empty:nil))))),
          list2(empty:nil),
          test(empty:nil),
          "action_start1",
          rule_bases(cons:
node_rule("start2",
          list1(cons:node_pattern1("start","yes",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil))))),
          list2(empty:nil),
          test(empty:nil),
```

```
          "action_start2",
          rule_bases(cons:
node_rule("start3",
          list1(cons:node_pattern1("start","yes",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))),
          list2(empty:nil),
          test(empty:nil),
          "action_start3",
          rule_bases(cons:
node_rule("end1",
          list1(empty:nil),
          list2(cons:node_pattern2(varcons(pos_const:0),"white",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"k"),"white",
               list2(empty:nil)))))))),
          test(test_id:"middle"),
          "action_end1",
          rule_bases(cons:
node_rule("end2",
          list1(empty:nil),
          list2(cons:node_pattern2(varcons(pos_const:0),"black",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"k"),"black",
               list2(empty:nil)))))))),
          test(test_id:"middle"),
          "action_end2",
          rule_bases(cons:
node_rule("user",
          list1(cons:node_pattern1("move","user",
               list1(empty:nil))),
          list2(empty:nil),
          test(empty:nil),
          "action_user",
          rule_bases(cons:
node_rule("move_to_putahi1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
               list2(empty:nil)))))),
          test(test_id:"adjacent"),
          "action_move_to_putahi1",
          rule_bases(cons:
node_rule("move_to_putahi2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
               list2(empty:nil)))))),
          test(test_id:"adjacent"),
          "action_move_to_putahi2",
          rule_bases(cons:
```

```
node_rule("move_from_putahi1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"white",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
               list2(empty:nil)))),
          test(empty:nil),
          "action_move_from_putahi1",
          rule_bases(cons:
node_rule("move_from_putahi2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"black",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
               list2(empty:nil)))),
          test(empty:nil),
          "action_move_from_putahi2",
          rule_bases(cons:
node_rule("move_adjacent1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"empty",
               list2(empty:nil)))),
          test(test_id:"adjacent"),
          "action_move_adjacent1",
          rule_bases(cons:
node_rule("move_adjacent2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"empty",
               list2(empty:nil)))),
          test(test_id:"adjacent"),
          "action_move_adjacent2",
          rule_bases(cons:
node_rule("dont_from_putahi1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"white",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"empty",
               list2(empty:nil)))))),
          test(test_id:"configuration0"),
          "action_dont",
          rule_bases(cons:
node_rule("dont_from_putahi2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"black",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"empty",
               list2(empty:nil)))))),
```

```
          test(test_id:"configuration0"),
          "action_dont",
          rule_bases(cons:
node_rule("dont_to_putahi1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"k"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"l"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"m"),"black",
               list2(empty:nil))))))))))))),
          test(test_id:"configuration1"),
          "action_dont",
          rule_bases(cons:
node_rule("dont_to_putahi2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"k"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"l"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"m"),"white",
               list2(empty:nil))))))))))))),
          test(test_id:"configuration1"),
          "action_dont",
          rule_bases(cons:
node_rule("do_to_putahi1",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","white",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"k"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"l"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"m"),"white",
               list2(empty:nil))))))))))))),
          test(test_id:"configuration2"),
          "action_do_to_putahi",
          rule_bases(cons:
node_rule("do_to_putahi2",
          list1(cons:node_pattern1("move","machine",
               list1(cons:node_pattern1("color","black",
               list1(empty:nil)))),
          list2(cons:node_pattern2(varcons(pos_const:0),"empty",
               list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
               list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"k"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"l"),"black",
               list2(cons:node_pattern2(varcons(pos_var:"m"),"black",
               list2(empty:nil))))))))))))),
          test(test_id:"configuration2"),
          "action_do_to_putahi",
          rule_bases(cons:
```

```
 node_rule("dont_adjacent1",
           list1(cons:node_pattern1("move","machine",
                list1(cons:node_pattern1("color","white",
                list1(empty:nil)))),
           list2(cons:node_pattern2(varcons(pos_const:0),"black",
                list2(cons:node_pattern2(varcons(pos_var:"i"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
                list2(cons:node_pattern2(varcons(pos_var:"k"),"black",
                list2(cons:node_pattern2(varcons(pos_var:"l"),"empty",
                list2(empty:nil))))))))))),
           test(test_id:"configuration3"),
           "action_dont",
           rule_bases(cons:
 node_rule("dont_adjacent2",
           list1(cons:node_pattern1("move","machine",
                list1(cons:node_pattern1("color","black",
                list1(empty:nil)))),
           list2(cons:node_pattern2(varcons(pos_const:0),"white",
                list2(cons:node_pattern2(varcons(pos_var:"i"),"black",
                list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"k"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"l"),"empty",
                list2(empty:nil))))))))))),
           test(test_id:"configuration3"),
           "action_dont",
           rule_bases(cons:
 node_rule("do_adjacent1",
           list1(cons:node_pattern1("move","machine",
                list1(cons:node_pattern1("color","white",
                list1(empty:nil)))),
           list2(cons:node_pattern2(varcons(pos_const:0),"black",
                list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
                list2(cons:node_pattern2(varcons(pos_var:"j"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"k"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"l"),"white",
                list2(cons:node_pattern2(varcons(pos_var:"m"),"white",
                list2(empty:nil))))))))))))),
           test(test_id:"configuration4"),
           "action_do_adjacent",
           rule_bases(cons:
 node_rule("do_adjacent2",
           list1(cons:node_pattern1("move","machine",
                list1(cons:node_pattern1("color","black",
                list1(empty:nil)))),
           list2(cons:node_pattern2(varcons(pos_const:0),"white",
                list2(cons:node_pattern2(varcons(pos_var:"i"),"empty",
                list2(cons:node_pattern2(varcons(pos_var:"j"),"black",
                list2(cons:node_pattern2(varcons(pos_var:"k"),"black",
                list2(cons:node_pattern2(varcons(pos_var:"l"),"black",
                list2(cons:node_pattern2(varcons(pos_var:"m"),"black",
                list2(empty:nil))))))))))))),
           test(test_id:"configuration4"),
           "action_do_adjacent",
       rule_bases(empty:nil))))))))))))
```

```
begin
 let fact_base := fact_bases("yes","","",boards(empty:nil))
 fact_base
end

!PRINTING OF THE BOARD
rec let print_board = proc(board:boards)
 if board isnt empty do
 begin
  writeString("BOARD ")
  writeInt(board'cons(position))
  writeString(board'cons(color)) ; writeString("'n")
  print_board(board'cons(next))
 end

!PRINTING OF THE STATE OF THE GAME
let print_state_of_game = proc(fb:fact_bases)
begin
 writeString("The state of the game is:'n")
 print_board(fb(board))                    !print board after each turn
 writeString("The next player is: ")
 writeString(fb(move)) ; writeString("'n")!print who's turn it is next
 writeString("Condition of start: ")
 writeString(fb(start)) ; writeString("'n")!print the condition of start
 writeString("The colour of the machine is: ")
 writeString(fb(color)) ; writeString("'n")!print colour of the machine
end

!PRINTING OF THE STATE OF THE GAME BEING LOGGED
rec let print_game = proc(game:games)
if game isnt empty then
begin
 print_state_of_game(game'cons(fact))
 print_game(game'cons(next))
end
else writeString("THE GAME BEING LOGGED IS EMPTY")

!SET OF PROCEDURES FOR THE RIGHT HAND SIDE OF THE RULES
let create_initial_board = proc( -> boards)
begin
 let board := boards(cons:
              node_boards(0,"empty",boards(cons:
              node_boards(1,"white",boards(cons:
              node_boards(2,"white",boards(cons:
              node_boards(3,"white",boards(cons:
              node_boards(4,"white",boards(cons:
              node_boards(5,"black",boards(cons:
              node_boards(6,"black",boards(cons:
              node_boards(7,"black",boards(cons:
              node_boards(8,"black",boards(empty:nil))))))))))))))))))
board
end

let ask_user_continue = proc( -> bool)
begin
 let result := false
 writeString("Do you want to continue?. Answer yes or no:'n")
 if readString() = "yes" then result := true
 else result := false
```

```
 result
end

let reinitialise_fact_base = proc(fb:fact_bases -> fact_bases)
begin
 fb(start) := "yes"
 fb(move) := ""
 fb(board) := boards(empty:nil)
 fb
end

let delete_fact_base = proc(fb:fact_bases -> fact_bases)
begin
 fb(start) := ""
 fb(move) := ""
 fb(color) := ""
 fb(board) := boards(empty:nil)
 fb
end

let machineWins = proc()
begin
writeString("The machine wins.'n")
end

let userWins = proc()
begin
writeString("The user wins.'n")
end

let modify_board = proc(b:boards ; p1,p2:int ; c1,c2:string -> boards)
begin
 !local procedure
rec let modify_position = proc(b:boards ; p:int ; c:string -> boards)
begin
 if b'cons(position) = p then b'cons(color) := c
 else b'cons(next) := modify_position(b'cons(next),p,c)
 b
end
!main body of procedure modify_board
 b := modify_position(b,p1,c1)
 b := modify_position(b,p2,c2)
 b
end

let action_start1 = proc(fb:fact_bases -> fact_bases)
begin
!local procedures
let ask_user_start = proc( -> bool)
begin
 let result := false
 writeString("Do you want to be the first to start the game?'n")
 writeString("Answer yes or no: ")
 if readString() = "yes" then result := true
 else result := false
 result
end
!main body of procedure action_start1
if read_game is empty then
```

```
begin
 fb(start) := ""
 fb(board) := create_initial_board()
 if ask_user_start() then {fb(color) := "white" ; fb(move) := "user"}
 else {fb(color) := "black" ; fb(move) := "machine"}
end
else fb := copy_state_game(read_game'cons(fact))
fb
end

let action_start2 = proc(fb:fact_bases -> fact_bases)
begin
 if read_game is empty then
 begin
  fb(start) := ""
  fb(move) := "machine"
  fb(color) := "black"
  fb(board) := create_initial_board()
 end
 else fb := copy_state_game(read_game'cons(fact))
 fb
end

let action_start3 = proc(fb:fact_bases -> fact_bases)
begin
 if read_game is empty then
 begin
  fb(start) := ""
  fb(move) := "user"
  fb(color) := "white"
  fb(board) := create_initial_board()
 end
 else fb := copy_state_game(read_game'cons(fact))
 fb
end

let action_end1 = proc(fb:fact_bases -> fact_bases)
begin
if fb(color) = "white" then machineWins()
else userWins()
if ask_user_continue() then fb := reinitialise_fact_base(fb)
else fb := delete_fact_base(fb)
fb
end

let action_end2 = proc(fb:fact_bases -> fact_bases)
begin
if fb(color) = "black" then machineWins()
else userWins()
if ask_user_continue() then fb := reinitialise_fact_base(fb)
else fb := delete_fact_base(fb)
fb
end

let action_user = proc(fb:fact_bases -> fact_bases)
begin
 let pos1 := 10 ; let pos2 :=10
 if read_move is empty then
 begin
```

```
   writeString("'n It is your turn to move. Indicate initial position: ")
   pos1 := readInt()
   writeString("'n Indicate final position: ")
   pos2 := readInt()
  end
  else
  begin
   if read_move'cons(who)="machine" do read_move := read_move'cons(next)
   pos1 := read_move'cons(pos1)
   pos2 := read_move'cons(pos2)
   writeString("NEXT MOVE READ FROM THE LOG_MOVE IS'n")
   writeString(read_move'cons(who))
   writeString("'n")
   writeInt(pos1)
   writeString("'n")
   writeInt(pos2)
   writeString("'n")
   writeString("END OF NEXT MOVE READ FROM THE LOG_MOVE IS'n")
   read_move := read_move'cons(next)
  end
  let c := ""
  case fb(color) of
      "white": c := "black"
      "black": c := "white"
      default: {}
  move := add_move(fb(move),pos1,pos2,move)
  fb(board) := modify_board(fb(board),pos1,pos2,"empty",c)
  fb(move) := "machine"
  fb
end

let action_move_to_putahi1 = proc(fb:fact_bases ; i:int -> fact_bases)
begin
  move := add_move(fb(move),i,0,move)
  fb(board) := modify_board(fb(board),0,i,"white","empty")
  fb(move) := "user"
  fb
end

let action_move_to_putahi2 = proc(fb:fact_bases ; i:int -> fact_bases)
begin
  move := add_move(fb(move),i,0,move)
  fb(board) := modify_board(fb(board),0,i,"black","empty")
  fb(move) := "user"
  fb
end

let action_move_from_putahi1 = proc(fb:fact_bases ; i:int -> fact_bases)
begin
  move := add_move(fb(move),0,i,move)
  fb(board) := modify_board(fb(board),0,i,"empty","white")
  fb(move) := "user"
  fb
end

let action_move_from_putahi2 = proc(fb:fact_bases ; i:int -> fact_bases)
begin
  move := add_move(fb(move),0,i,move)
  fb(board) := modify_board(fb(board),0,i,"empty","black")
```

```
 fb(move) := "user"
 fb
end

let action_move_adjacent1 = proc(fb:fact_bases ; i,j:int -> fact_bases)
begin
 move := add_move(fb(move),i,j,move)
 fb(board) := modify_board(fb(board),i,j,"empty","white")
 fb(move) := "user"
 fb
end

let action_move_adjacent2 = proc(fb:fact_bases ; i,j:int -> fact_bases)
begin
 move := add_move(fb(move),i,j,move)
 fb(board) := modify_board(fb(board),i,j,"empty","black")
 fb(move) := "user"
 fb
end

!ENGINE
!PATTERN MATCHING OF EACH PATTERN OF A RULE
let pattern_matching = proc(arg:argument_pm ; fb:fact_bases -> results_matching)
begin
!local procedures
let put_facts_in_wm = proc(fn:string ; wm:wms ; fb:fact_bases -> wms)
begin
!local procedure
rec let create_board_wm = proc(bb,bb1:boards -> boards)
begin
 if bb is empty then bb1 := bb1
 else bb1 := boards(cons:node_boards(bb'cons(position),bb'cons(color),create_board_wm(bb'cons(next),bb1)))
 bb1
end
 case fn of
     "start": wm := wms(start:fb(start))
     "color": wm := wms(color:fb(color))
     "move" : wm := wms(move:fb(move))
     "board": wm := wms(board:create_board_wm(fb(board),boards(empty:nil)))
     default: {}
 wm
end

let print_wm = proc(wm:wms)
begin
 if wm is start do {writeString(wm'start) ; writeString("'n")}
 if wm is color do {writeString(wm'color) ; writeString("'n")}
 if wm is move do {writeString(wm'move) ; writeString("'n")}
 if wm is board do {print_board(wm'board) ; writeString("was board'n")}
end

let match_constants = proc(wm:wms ; arg:argument_pm -> wms)
begin
!local procedures
let compare_pos = proc(wm:wms ; pos:int -> wms)
begin
 while wm'board'cons(position) ~= pos do
 begin
  wm := wms(board:wm'board'cons(next))
```

```
 end
 wm'board'cons(next) := boards(empty:nil)
 wm
end
let compare_color = proc(wm:wms ; color:string -> wms)
begin
 let aux := boards(empty:nil)
 while wm'board isnt empty do
 begin
  if wm'board'cons(color) = color do
   aux := boards(cons:node_boards(wm'board'cons(position),wm'board'cons(color),aux))
  wm := wms(board:wm'board'cons(next))
 end
 if aux is empty then wm := wms(empty:nil)
 else wm := wms(board:aux)
 wm
end
!main body of match_constants
 if wm is start do
  if arg'pattern1(fact_value) ~= wm'start do wm := wms(empty:nil)
 if wm is color do
  if arg'pattern1(fact_value) ~= wm'color do wm := wms(empty:nil)
 if wm is move do
  if arg'pattern1(fact_value) ~= wm'move do wm := wms(empty:nil)
 if wm is board do
 begin
  if arg'pattern2(position) is pos_const do
   wm := compare_pos(wm,arg'pattern2(position)'pos_const)
  wm := compare_color(wm,arg'pattern2(color))
 end
 wm
end

let match_variables = proc(wm:wms ; bb:node_pattern2 -> singles)
begin
 let list_of_values := singles("",list3(empty:nil))
 if bb(position) is pos_var do
 begin
  list_of_values(var_pos) := bb(position)'pos_var
  while wm'board isnt empty do
  begin
   list_of_values(pos_list):=list3(cons:node_pos(wm'board'cons(position),list_of_values(pos_list)))
   wm := wms(board:wm'board'cons(next))
  end
 end
 list_of_values
end
!main body of the procedure pattern_matching
 let wm := wms(empty:nil)
 let result := results_matching(without_var:true)
 let list_of_values := singles("",list3(empty:nil))
 if arg is pattern1 then
 begin
  let fn := arg'pattern1(fact_name)
  wm := put_facts_in_wm(fn,wm,fb)
  wm := match_constants(wm,arg)
  if wm is empty do result := results_matching(without_var:false)
 end
 else
```

```
 begin
  wm := put_facts_in_wm("board",wm,fb)
  wm := match_constants(wm,arg)
  if wm is empty then result := results_matching(without_var:false)
  else
  begin
   list_of_values := match_variables(wm,arg'pattern2)
   if list_of_values(var_pos) ~= "" do
    if list_of_values(pos_list) is empty then
     result := results_matching(without_var:false)
    else result := results_matching(with_var:list_of_values)
  end
 end
 result
end

!APPEND PROCEDURES
rec let append_singles = proc(pr:patterns_results ; r:singles -> patterns_results)
begin
 if pr is empty then pr := patterns_results(cons:node_results(r,pr))
 else pr'cons(next) := append_singles(pr'cons(next),r)
 pr
end

rec let append_agenda = proc(ag,final_list:agendas -> agendas)
begin
 if ag is empty then ag := final_list
 else ag'cons(next) := append_agenda(ag'cons(next),final_list)
 ag
end

!AUXILIAR PROCEDURES TO FORM THE ELEMENTS OF THE AGENDA
!concl4,form_tuples,do_tuples, and sub_tuples form tuples
rec let concl4 = proc(l1,l2:list4 -> list4)
 project l1 as L onto
 cons: list4(cons:node_tuples(L(position1),L(position2),concl4(L(next),l2)))
 default: l2

rec let sub_tuples = proc(x:int ; l2:list3 -> list4)
 project l2 as L onto
 cons: list4(cons:node_tuples(x,L(position),sub_tuples(x,L(next))))
 default: list4(empty:nil)

rec let do_tuples = proc(l1,l2:list3 -> list4)
 project l1 as L onto
 cons: concl4(sub_tuples(L(position),l2),do_tuples(L(next),l2))
 default: list4(empty:nil)

let form_tuples = proc(s1,s2:singles -> tuples)
 tuples(s1(var_pos),s2(var_pos),do_tuples(s1(pos_list),s2(pos_list)))

!concl5,sub_trip2,sub_trip1,do_triples, and form_triples form triples
rec let concl5 = proc(l1,l2:list5 -> list5)
 project l1 as L onto
 cons: list5(cons:node_triples(L(position1),L(position2),L(position3),concl5(L(next),l2)))
 default: l2

rec let sub_trip2 = proc(x,y:int ; l:list3 -> list5)
 project l as L onto
```

```
 cons: list5(cons:node_triples(x,y,L(position),sub_trip2(x,y,L(next))))
 default: list5(empty:nil)

rec let sub_trip1 = proc(x:int ; l2,l3:list3 -> list5)
 project l2 as L onto
 cons: concl5(sub_trip2(x,L(position),l3),sub_trip1(x,L(next),l3))
 default: list5(empty:nil)

rec let do_triples = proc(l1,l2,l3:list3 -> list5)
 project l1 as L onto
 cons: concl5(sub_trip1(L(position),l2,l3),do_triples(L(next),l2,l3))
 default: list5(empty:nil)

let form_triples = proc(s1,s2,s3:singles -> triples)
 triples(s1(var_pos),s2(var_pos),s3(var_pos),do_triples(s1(pos_list),s2(pos_list),s3(pos_list)))

!FUNCTIONS OF THE LEFT HAND SIDE OF THE RULES
!adjacent indicates if two positions of the board are adjacent or not
let adjacent = proc(i,j:int -> bool)
begin
 let result := false
 if i=0 or j=0 then result := false
 else
  case i of
          1: if j=8 or j=2 then result := true
             else result := false
          8: if j=1 or j=7 then result := true
             else result := false
     default: if abs(i-j)=1 then result := true
              else result := false
 result
end

!middle indicates if a position i is in between the positions j and k
let middle = proc(i,j,k:int -> bool)
begin
 let result := false
 if i=0 or j=0 or k=0 then result := false
 else
  case i of
          1: if (j=8 and k=2) or (j=2 and k=8) then result := true
             else result := false
          8: if (j=1 and k=7) or (j=7 and k=1) then result := true
             else result := false
     default: if (j=(i+1) and k=(i-1)) or (j=(i-1) and k=(i+1))
              then result := true
              else result := false
 result
end


!functions for the lhs of the rules applied to the list of possibilities
rec let is_inl = proc(i:int ; l:list4 -> bool)
 project l as L onto
  cons: L(position1)=i or is_inl(i,L(next))
  default: false

rec let not_repeated = proc(l:list4 -> list4)
 project l as L onto
```

```
 cons: if is_inl(L(position1),L(next)) then not_repeated(L(next))
       else list4(cons:node_tuples(L(position1),L(position2),not_repeated(L(next))))
 default:list4(empty:nil)

rec let adjacent_of_list = proc(l1,l2:list4 -> list4)
begin
 if l1 is empty then l2 := l2
 else
 begin
  let i := l1'cons(position1)
  let j := l1'cons(position2)
  if adjacent(i,j) do l2 := list4(cons:node_tuples(i,j,l2))
  l2 := adjacent_of_list(l1'cons(next),l2)
 end
 l2
end

rec let middle_of_list = proc(l1,l2:list5 -> list5)
begin
 if l1 is empty then l2 := l2
 else
 begin
  let i := l1'cons(position1)
  let j := l1'cons(position2)
  let k := l1'cons(position3)
  if middle(i,j,k) do l2 := list5(cons:node_triples(i,j,k,l2))
  l2 := middle_of_list(l1'cons(next),l2)
 end
 l2
end

!AUXILIAR PROCEDURES FOR THE STRATEGIC RULES
!Is_strategic looks if a given rule is strategic or legal move
let is_strategic = proc(id:string -> bool)
 case id of
  "dont_from_putahi1","dont_from_putahi2","dont_to_putahi1",
  "dont_to_putahi2","do_to_putahi1","do_to_putahi2","dont_adjacent1",
  "dont_adjacent2","do_adjacent1","do_adjacent2": true
  default: false

! These are auxiliar procedures for the configurations
!isk_inl looks is an integer is in a list
rec let isk_inl = proc(k:int ; l:list5 -> bool)
 project l as L onto
  cons: L(position2)=k or isk_inl(k,L(next))
  default: false

!isi_inl looks if an integer is in a list
rec let isi_inl = proc(i:int ; l:list5 -> bool)
 project l as L onto
  cons: L(position3)=i or isi_inl(i,L(next))
  default: false

!get_i gets the i in l1 that has its k in l2
rec let get_i = proc(l1,l2:list5 -> int)
 project l1 as L onto
 cons:if isk_inl(L(position3),l2) then L(position1)
      else get_i(L(next),l2)
 default: 10
```

```
!get_i2 gets the i that has l as adjacent
rec let get_i2 = proc(l1:list5 ; x:int -> int)
 project l1 as L onto
 cons:if adjacent(L(position3),x) then L(position3)
      else get_i2(L(next),x)
 default: 10

!two_is gets the i that is twice in a list
rec let two_is = proc(l:list5 -> int)
 project l as L onto
 cons:if isi_inl(L(position3),L(next)) then L(position3)
      else two_is(L(next))
 default: 10

!get_k gets the k that is next to empty and another of its colour
rec let get_k = proc(l1,l2:list5 -> int)
 project l1 as L onto
 cons:if isk_inl(L(position3),l2) then L(position3)
      else get_k(L(next),l2)
 default: 10

!get_ks finds the two adjacent positions to a given one
let get_ks = proc(l:list5 -> list4)
begin
 let i := two_is(l)
 if i ~=10 then
 begin
  while l'cons(position3)~=i do l := l'cons(next)
  let k1 := l'cons(position1) ; l := l'cons(next)
  while l'cons(position3)~=i do l := l'cons(next)
  let k2 := l'cons(position1)
  list4(cons:node_tuples(k1,k2,list4(empty:nil)))
 end
 else list4(empty:nil)
end

!PROCEDURES TO TEST THE STRATEGIC CONFIGURATIONS
!configuration0 tests for the configuration in which the machine
!can move from putahi or adjacent:putahi=m & m_
let configuration0 = proc(list:patterns_results -> elem_type)
begin
 let done := false ; let i := list3(empty:nil)
 let j := list3(empty:nil)
 while ~done do
  project list as L onto
  cons:
       begin
        case L(results)(var_pos) of
         "i": i := L(results)(pos_list)
         "j": j := L(results)(pos_list)
         default: writeString("Error in Configuration0")
        list:= L(next)
       end
 default: done := true
 let l := do_tuples(i,j)
 l := adjacent_of_list(l,list4(empty:nil))
 if l is empty then  elem_type(elem0:false)
 else elem_type(elem1:singles("p",list3(cons:node_pos(0,list3(empty:nil)))))
```

```
end

!configuration1 tests for the configuration umuuu
let configuration1 = proc(list:patterns_results -> elem_type)
begin
 let done := false ; let i := list3(empty:nil)
 let j := list3(empty:nil) ; let k := list3(empty:nil)
 let l := list3(empty:nil) ; let m := list3(empty:nil)
 while ~done do
  project list as L onto
  cons:
       begin
        case L(results)(var_pos) of
         "i": i := L(results)(pos_list)
         "j": j := L(results)(pos_list)
         "k": k := L(results)(pos_list)
         "l": l := L(results)(pos_list)
         "m": m := L(results)(pos_list)
         default: writeString("Error in Configuration1")
        list:= L(next)
       end
  default: done := true
 let l1 := do_triples(i,j,k)
 let l2 := do_triples(l,k,m)
 l1 := middle_of_list(l1,list5(empty:nil))
 l2 := middle_of_list(l2,list5(empty:nil))
 let i1 := get_i(l1,l2)
 if i1~=10 then elem_type(elem1:singles("i",list3(cons:node_pos(i1,list3(empty:nil)))))
 else elem_type(elem0:false)
end

!configuration2 tests for the configuration mmumm
let configuration2 = proc(list:patterns_results -> elem_type)
begin
 let done := false ; let i := list3(empty:nil)
 let j := list3(empty:nil) ; let k := list3(empty:nil)
 while ~done do
  project list as L onto
  cons:
       begin
        case L(results)(var_pos) of
         "i": i := L(results)(pos_list)
         "j": j := L(results)(pos_list)
         "k": k := L(results)(pos_list)
         default: writeString(L(results)(var_pos))
        list:= L(next)
       end
  default: done := true
 let l1 := do_triples(k,j,i)
 l1 := middle_of_list(l1,list5(empty:nil))
 let ks := get_ks(l1)
 project ks as L onto
 cons: elem_type(elem2:tuples("j","k",ks))
 default: elem_type(elem0:false)
end

!configuration3 tests for the configuration uum_
let configuration3 = proc(list:patterns_results -> elem_type)
begin
```

```
 let done := false ; let i := list3(empty:nil)
 let j := list3(empty:nil) ; let k := list3(empty:nil)
 let l := 10
 while ~done do
  project list as L onto
  cons:
       begin
        case L(results)(var_pos) of
         "i": i := L(results)(pos_list)
         "j": j := L(results)(pos_list)
         "k": k := L(results)(pos_list)
         "l": l := L(results)(pos_list)'cons(position)
         default: writeString("Error in Configuration3")
        list:= L(next)
       end
  default: done := true
 let l1 := do_triples(k,j,i)
 l1 := middle_of_list(l1,list5(empty:nil))
 let i1 := get_i2(l1,l)
 if i1~=10 then elem_type(elem1:singles("i",list3(cons:node_pos(i1,list3(empty:nil)))))
 else elem_type(elem0:false)
end

!configuration4 tests for the configuration mmm_m
let configuration4 = proc(list:patterns_results -> elem_type)
begin
 let done := false ; let i := list3(empty:nil)
 let j := list3(empty:nil) ; let k := list3(empty:nil)
 let l := list3(empty:nil) ; let m := list3(empty:nil)
 while ~done do
  project list as L onto
  cons:
       begin
        case L(results)(var_pos) of
         "i": i := L(results)(pos_list)
         "j": j := L(results)(pos_list)
         "k": k := L(results)(pos_list)
         "l": l := L(results)(pos_list)
         "m": m := L(results)(pos_list)
         default: writeString("Error in Configuration4")
        list:= L(next)
       end
  default: done := true
 let l1 := do_triples(i,j,k)
 let l2 := do_triples(l,k,m)
 l1 := middle_of_list(l1,list5(empty:nil))
 l2 := middle_of_list(l2,list5(empty:nil))
 let k1 := get_k(l1,l2)
 if k1~=10 then elem_type(elem1:singles("k",list3(cons:node_pos(k1,list3(empty:nil)))))
 else elem_type(elem0:false)
end

!Treat_strategic_rule creates the strategic elements of the agenda
let treat_strategic_rule = proc(ag:agendas ; rule:node_rule ;l:patterns_results -> agendas)
begin
let move := elem_type(elem0:false)
let c := rule(pattern3)'test_id
case c of
 "configuration0": move := configuration0(l)
```

```
 "configuration1": move := configuration1(1)
 "configuration2": move := configuration2(1)
 "configuration3": move := configuration3(1)
 "configuration4": move := configuration4(1)
          default: {}
if move is elem0 then ag
else append_agenda(ag,agendas(cons:node_agenda(rule(rule_id),move,agendas(empty:nil))))
end

!AUXILIAR PROCEDURES FOR THE ACTIONS OF THE STRATEGIC RULES
!many_moves looks if a rule in the agenda has more than one move
rec let many_moves = proc(a:agendas ; id:string -> bool)
 project a as A onto
 cons:if A(rule_id)=id then
       project A(agenda_elem) as AA onto
       elem2:project AA(tuples_list) as AAA onto
             cons: if AAA(next) isnt empty then true else false
             default:{writeString("Empty move in agenda element");false}
       default: false
      else many_moves(A(next),id)
 default: {writeString("Error: agenda empty") ; false}

!legals gives the number of legal move rules in the agenda
rec let legals = proc(a:agendas -> int)
 project a as A onto
 cons: if is_strategic(A(rule_id)) then legals(A(next))
       else legals(A(next)) + 1
 default: 0

!delete1 deletes a move from a list of moves
rec let delete1 = proc(l:list4 ; i:int -> list4)
 project l as L onto
 cons: if L(position1) = i then delete1(L(next),i)
       else list4(cons:node_tuples(L(position1),L(position2),delete1(L(next),i)))
 default: list4(empty:nil)

!delete2 deletes all moves from a list of moves except the one given
rec let delete2 = proc(l:list4 ; i,j:int -> list4)
 project l as L onto
 cons: if L(position1)~=i and L(position1)~=j then delete2(L(next),i,j)
       else list4(cons:node_tuples(L(position1),L(position2),delete2(L(next),i,j)))
 default: list4(empty:nil)

!delete3 deletes all moves from a list of moves except the one given
rec let delete3 = proc(l:list4 ; i:int -> list4)
 project l as L onto
 cons: if L(position1)~=i then delete3(L(next),i)
       else list4(cons:node_tuples(L(position1),L(position2),delete3(L(next),i)))
 default: list4(empty:nil)

!delete_move1 deletes the move i from the rule id in the agenda a
rec let delete_move1 = proc(a:agendas ; id:string ; i:int -> agendas)
 project a as A onto
 cons: if A(rule_id)~=id then
        agendas(cons:node_agenda(A(rule_id),A(agenda_elem),delete_move1(A(next),id,i)))
       else
        agendas(cons:node_agenda(A(rule_id),project A(agenda_elem) as AA onto
         elem2: elem_type(elem2:tuples(AA(var_pos1),AA(var_pos2),delete1(AA(tuples_list),i)))
                          default:{writeString("bad agenda elem") ; elem_type(elem0:false)},
```

```
                              delete_move1(A(next),id,i)))
 default: agendas(empty:nil)

!delete_move2 deletes all moves except i,j from the rule id in the agenda a
rec let delete_move2 = proc(a:agendas ; id:string ; i,j:int -> agendas)
 project a as A onto
 cons: if A(rule_id)~=id then
        agendas(cons:node_agenda(A(rule_id),A(agenda_elem),delete_move2(A(next),id,i,j)))
       else
        agendas(cons:node_agenda(A(rule_id),
                          project A(agenda_elem) as AA onto
                           elem2: elem_type(elem2:
                                 tuples(AA(var_pos1),AA(var_pos2),delete2(AA(tuples_list),i,j)))
                           default:{writeString("bad agenda elem") ; elem_type(elem0:false)},
                           delete_move2(A(next),id,i,j)))
 default: agendas(empty:nil)

!delete_move2 deletes all moves except i,j from the rule id in the agenda
rec let delete_move3 = proc(a:agendas ; id:string ; i:int -> agendas)
 project a as A onto
 cons: if A(rule_id)~=id then
        agendas(cons:node_agenda(A(rule_id),A(agenda_elem),delete_move3(A(next),id,i)))
       else
        agendas(cons:node_agenda(A(rule_id),
                          project A(agenda_elem) as AA onto
                          elem2: elem_type(elem2:tuples(AA(var_pos1),AA(var_pos2),
                                                 delete3(AA(tuples_list),i)))
                          default:{writeString("bad agenda elem") ; elem_type(elem0:false)},
                          delete_move3(A(next),id,i)))
 default: agendas(empty:nil)

rec let delete_rule = proc(a:agendas ; id:string -> agendas) !delete_rule deletes a rule from the agenda
 project a as A onto
 cons: if A(rule_id)~=id then
        agendas(cons:node_agenda(A(rule_id),A(agenda_elem),delete_rule(A(next),id)))
       else delete_rule(A(next),id)
 default: agendas(empty:nil)

!ACTIONS OF THE STRATEGIC RULES
!action_dont deletes bad moves
let action_dont = proc(a:agendas ; id1,id2:string ; i:int -> agendas)
 delete_rule(if many_moves(a,id1) then delete_move1(a,id1,i)
             else if legals(a)>1 then delete_rule(a,id1) else a,id2)

!action_do_to_putahi removes the moves that are not the best two
let action_do_to_putahi = proc(a:agendas ; id1,id2:string ; i,j:int -> agendas)
 delete_rule(delete_move2(a,id1,i,j),id2)

!action_do_adjacent removes the moves that are not the one best
let action_do_adjacent = proc(a:agendas ; id1,id2:string ; i:int -> agendas)
 delete_rule(delete_move3(a,id1,i),id2)

!DETECTION OF RULES THAT ACCOMPLISH THE FACT BASE
let detection = proc(rb:rule_bases ; fb:fact_bases -> agendas)
begin   !main body of detection
 let agenda := agendas(empty:nil)
 while rb isnt empty do
 begin
 let rule := node_rule(rb'cons(rule_id),
```

```
                    rb'cons(pattern1),
                    rb'cons(pattern2),
                    rb'cons(pattern3),
                    rb'cons(right_side),
                    rule_bases(empty:nil))
let abandon_rule := false
let list_of_results := patterns_results(empty:nil)
let final_list := agendas(empty:nil)
let list_pattern1 := rule(pattern1)
let list_pattern2 := rule(pattern2)
while list_pattern1 isnt empty and ~abandon_rule do
begin
 let result :=
  pattern_matching(argument_pm(pattern1:list_pattern1'cons),fb)
 if result'without_var=false do abandon_rule := true
 list_pattern1 := list_pattern1'cons(next)
end
while list_pattern2 isnt empty and ~abandon_rule do
begin
 if fb(board) is empty then abandon_rule := true
 else
 begin
  let result :=
   pattern_matching(argument_pm(pattern2:list_pattern2'cons),fb)
  if result is with_var then
   list_of_results := append_singles(list_of_results,result'with_var)
  else
   if result'without_var=false do abandon_rule := true
  list_pattern2 := list_pattern2'cons(next)
 end
end
 if ~abandon_rule do
  if is_strategic(rule(rule_id)) then        !it is a strategic rule
   agenda := treat_strategic_rule(agenda,rule,list_of_results)
  else                                       !it is a legal move rule
 begin
  if list_of_results is empty then
  begin
   final_list := agendas(cons:node_agenda(rule(rule_id),elem_type(elem0:true),
                                          agendas(empty:nil)))
   agenda := append_agenda(agenda,final_list)
  end
  else
  begin
   let l := list_of_results'cons
   if l(next) is empty then
   begin
    final_list := agendas(cons:node_agenda(rule(rule_id),elem_type(elem1:l(results)),
                                           agendas(empty:nil)))
    agenda := append_agenda(agenda,final_list)
   end
   else
   begin
    let ll := l(next)'cons
    if ll(next) is empty then
    begin
     final_list := agendas(cons:node_agenda(rule(rule_id),
                                elem_type(elem2:form_tuples(l(results),ll(results))),
                                          agendas(empty:nil)))
```

```
      final_list'cons(agenda_elem)'elem2(tuples_list) := not_repeated(adjacent_of_list(
           final_list'cons(agenda_elem)'elem2(tuples_list),list4(empty:nil)))
      if final_list'cons(agenda_elem)'elem2(tuples_list) isnt empty do
       agenda := append_agenda(agenda,final_list)
     end
     else
     begin
      let lll := ll(next)'cons
      final_list := agendas(cons:node_agenda(rule(rule_id),
                            elem_type(elem3:form_triples(l(results),ll(results),lll(results))),
                                      agendas(empty:nil)))
      final_list'cons(agenda_elem)'elem3(triples_list) :=
       middle_of_list(final_list'cons(agenda_elem)'elem3(triples_list),list5(empty:nil))
      if final_list'cons(agenda_elem)'elem3(triples_list) isnt empty do
       agenda := append_agenda(agenda,final_list)
     end
    end
   end
  end
  rb := rb'cons(next)
 end
 agenda
end

!CHOICE OF THE RULE TO BE FIRED BETWEEN ALL RULES DETECTED AS GOOD
!Treat_agenda applies the strategic rules to the legal rules so that the
!agenda is left with one good legal rule with good moves in it
rec let treat_agenda = proc(a,a1:agendas -> agendas)
 project a1 as A onto
 cons:if ~is_strategic(A(rule_id)) then treat_agenda(a,A(next))
      else
      begin
       case A(rule_id) of
        "dont_from_putahi1":
          treat_agenda(action_dont(a,"move_from_putahi","dont_from_putahi1",
                       A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        "dont_from_putahi2":
         treat_agenda(action_dont(a,"move_from_putahi2","dont_from_putahi2",
                       A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        "dont_to_putahi1":
          treat_agenda(action_dont(a,"move_to_putahi1","dont_to_putahi1",
                       A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        "dont_to_putahi2":
          treat_agenda(action_dont(a,"move_to_putahi2","dont_to_putahi2",
                       A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        "dont_adjacent1":
          treat_agenda(action_dont(a,"move_adjacent1","dont_adjacent1",
                       A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        "dont_adjacent2":
          treat_agenda(action_dont(a,"move_adjacent2","dont_adjacent2",
                       A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        "do_to_putahi1":
          treat_agenda(action_do_to_putahi(a,"move_to_putahi1","do_to_putahi1",
                    A(agenda_elem)'elem2(tuples_list)'cons(position1),
                    A(agenda_elem)'elem2(tuples_list)'cons(position2)),A(next))
        "do_to_putahi2":
          treat_agenda(action_do_to_putahi(a,"move_to_putahi2","do_to_putahi2",
                    A(agenda_elem)'elem2(tuples_list)'cons(position1),
                    A(agenda_elem)'elem2(tuples_list)'cons(position2)),A(next))
```

```
        "do_adjacent1":
          treat_agenda(action_do_adjacent(a,"move_adjacent1","do_adjacent1",
                    A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        "do_adjacent2":
          treat_agenda(action_do_adjacent(a,"move_adjacent2","do_adjacent2",
                    A(agenda_elem)'elem1(pos_list)'cons(position)),A(next))
        default: agendas(empty:nil)
      end
 default: a

!choose_move chooses ramdomly between the different good moves of a rule and puts it in front
let choose_move = proc(a:elem_type -> elem_type)
 project a as A onto
 elem2:project A(tuples_list) as AA onto
    cons:project AA(next) as AAA onto
         cons:project AAA(next) as AAAA onto
              cons:project AAAA(next) as AAAAA onto
                   cons: if (time() rem 2)=0 then
                          elem_type(elem2:tuples(A(var_pos1),A(var_pos2),AA(next)))
                         else
                          if ((time()+1) rem 2)=0 then
                           elem_type(elem2:tuples(A(var_pos1),A(var_pos2),AAA(next)))
                          else
                           if ((time()+5) rem 2)=0 then
                            elem_type(elem2:tuples(A(var_pos1),A(var_pos2),AAAA(next)))
                           else a
                   default: if (time() rem 2)=0 then
                             elem_type(elem2:tuples(A(var_pos1),A(var_pos2),AA(next)))
                            else
                             if ((time()+1) rem 2)=0 then
                              elem_type(elem2:tuples(A(var_pos1),A(var_pos2),AAA(next)))
                             else a
              default: if (time() rem 2)=0 then
                        elem_type(elem2:tuples(A(var_pos1),A(var_pos2),AA(next)))
                       else a
         default: a
    default: a
 default: a

let choice = proc(ag:agendas -> node_agenda) !CHOICE CHOOSES THE FIRST MOVE IN THE AGENDA
begin
 ag := treat_agenda(ag,ag)
 print_agenda(ag)
 node_agenda(ag'cons(rule_id),ag'cons(agenda_elem),agendas(empty:nil))
end

let deduction = proc(na:node_agenda ; fb:fact_bases -> fact_bases) !EXECUTION OF THE RULE CHOSEN
begin
 print_elem(na(agenda_elem))
 let elem := na(agenda_elem)
 if elem is elem0 do
  case na(rule_id) of
      "start1": fb := action_start1(fb)
      "start2": fb := action_start2(fb)
      "start3": fb := action_start3(fb)
        "user": fb := action_user(fb)
       default:{}
 if elem is elem1 do
 begin
```

```
  let i := elem'elem1(pos_list)'cons(position)
  case na(rule_id) of
      "move_from_putahi1": fb := action_move_from_putahi1(fb,i)
      "move_from_putahi2": fb := action_move_from_putahi2(fb,i)
                  default:{}
 end
 if elem is elem2 do
 begin                              !it assumes that the order is (i,j)
  let i := elem'elem2(tuples_list)'cons(position1)
  let j := elem'elem2(tuples_list)'cons(position2)
  if elem'elem2(var_pos1) = "j" and elem'elem2(var_pos2) = "i" do
  { let p := i ; i := j ; j := p} ! it happens that the order is (j,i)
  case na(rule_id) of
      "move_to_putahi1": fb := action_move_to_putahi1(fb,i)
      "move_to_putahi2": fb := action_move_to_putahi2(fb,i)
       "move_adjacent1": fb := action_move_adjacent1(fb,i,j)
       "move_adjacent2": fb := action_move_adjacent2(fb,i,j)
              default:{}
 end
 if elem is elem3 do
  case na(rule_id) of
      "end1": fb := action_end1(fb)
      "end2": fb := action_end2(fb)
     default:{}
 store_game(na,fb)
 print_state_of_game(fb)
 fb
end

!engine is the engine of the rule base system
let engine = proc(rule_base:rule_bases ; fact_base:fact_bases)
begin
 let finish := false
 while ~finish do
 begin
  let agenda := detection(rule_base,fact_base)
  print_agenda(agenda)
  if agenda is empty then finish := true
  else
  begin
   let chosen_rule := choice(agenda)
   log_system(chosen_rule)
   fact_base := deduction(chosen_rule,fact_base)
  end
 end
end

!MAIN BODY OF PROGRAM
 let rule_base := rule_bases(empty:nil)
 rule_base := create_rules(rule_base)   ! print_rules(rule_base)
 let fact_base := initialise_game()
 engine(rule_base,fact_base)
end
```

E.3 Final Program for Mu Torere

The final program is the result of applying the transformation process to both prototype programs, and it is part in the original file, and part in the persistent store. For brevity, here I will consider that the final program is constituted by what is left in the original file, plus the persistent operations that are directly used in this file. These persistent operations make use of others, and I will give an example of this use with the case of the operation that creates one of the rules of the rule base.

E.3.1 Part of the Final Program in Original File

```
!TYPES OF THE FACT BASE
!colour contains the colour the program is playing with:none,white or black
!move indicates whose turn is to move, the program or the user
!start indicates if it is the begining of a game or not: yes,no
!boards is the type of the state of the board in the game
!struct2 is the type of the fact base
 type struct2 is struct2     ! & (list1 & struct1)
!TYPES OF RULE BASE
!type of pattern1
 !type list3 is variant3 !& struct3
!types for pattern2
 !type list4 is list4 ! & struct4 (& variant2)
 !type variant2 is variant2
!type of pattern3
 !type variant7 is variant7
!list5 is the type of the rule base
 type list5 is list5 ! & struct5 (& list3 & list4 & variant7)
!TYPES OF WORKING MEMORY
!variant9 is the type of the working memory
 !type variant9 is variant9  ! & list1
!TYPES OF THE AGENDA
!list12 is the type of the input elements for the agenda
 !type list12 is list12 ! & struct12 (& variant11)
!variant11 is the type of the elements of the agenda
 !type variant11 is variant11 ! & (struct7 & struct9 & struct11)
  !type struct7 is struct7     ! & (list6 & struct6)
  !type struct9 is struct9     ! & (list8 & struct8)
  !type struct11 is struct11   ! & (list10 & struct10)
!OTHER TYPES FOR THE PROGRAM
!list13 is the type of the element containing all the variable
!results of a rule
!type list13 is list13 ! & struct13 (& struct7)
!variant18 is the type of the argument to the pattern matching function
 !type variant18 is variant18 ! & struct3 & struct4
!variant19 is the type of the result from the pattern matching of a
!pattern
 !type variant19 is variant19  ! & struct7
```

```
!TYPES FOR THE LOG SYSTEM
!type list14 is variant14 ! & struct14(& struct2(& variant1(& struct1)))
!type list15 is variant15 ! & struct15
!type list16 is variant16 ! & struct16(& variant14(
!                                      & struct14(
!                                      & struct2(& variant1(& struct1)))))
!type list17 is variant17 ! & struct17(& variant15(& struct15))
!DECLARATION OF PERSISTENT ENVIRONMENTS
let ps = PS()
use ps with User:env in
use User with torere_env:env in
use torere_env with analysis1_env,analysis2_env:env in
use analysis1_env with level2_env:env in
use level2_env with substructures_env:env in
use analysis2_env with level22_env:env in
use level22_env with list2_env:env in
begin
!MAIN BODY OF PROGRAM
let rule_base := use list2_env with create_list5:proc( -> list5) in
 create_list5()     ! print_list5(rule_base)
let fact_base := use substructures_env with create_struct2:proc( -> struct2)
 in create_struct2()
use list2_env with treat_15_s2:proc(list5,struct2) in
 treat_15_s2(rule_base,fact_base)
end
```

E.3.2 Part of Program in Persistent Store

Here I present the operations made persistent during the transformation process, and which are directly used in the program left in the file: `create_list5` (create the rule base), `print_list5` (print the rule base), `create_struct2` (create the initial fact base), and `treat_15_s2` (the engine). These are only some of the operations that are in the persistent store, and we can see that they make use of other persistent operations, e.g., `create_list5` makes use of all the `creates5_i` operations. I illustrate how the operation `creates5_a` used in `create_list5`, uses other operations, which also use other operations, until all the operations involved in the functionality of `creates5_a` are shown.

Some operations were made persistent during the analysis of prototype one, and used from the environments created then, e.g., `creates5_a...creates5_l` which create the rules of the first rule base are used from the environment `structs_of_lists_env`. Other operations were made persistent during the analysis of prototype two, e.g., `creates5_m...creates5_v` (strategic rules) which are used from the environment `structs_of_lists2_env`. Finally, other

operations, e.g., `create_list5` (creation of the rule base) were made persistent in the analysis of prototype one, and subsequently changed and made persistent again in the analysis of prototype two (in this case to incorporate the strategic rules); the `create_list5` shown here is the second one.

```
let create_list5 = proc( -> list5) !CREATE_LIST5 CREATES THE RULE BASE OF PROTOTYPE TWO
use list_env with add5:proc(list5,struct5 -> list5) ; empty5:proc( -> list5) in
use structs_of_lists_env with
 creates5_a:proc( -> struct5) ; creates5_b:proc( -> struct5) ; creates5_c:proc( -> struct5) ;
 creates5_d:proc( -> struct5) ; creates5_e:proc( -> struct5) ; creates5_f:proc( -> struct5) ;
 creates5_g:proc( -> struct5) ; creates5_h:proc( -> struct5) ; creates5_i:proc( -> struct5) ;
creates5_j:proc( -> struct5) ; creates5_k:proc( -> struct5) ; creates5_l:proc( -> struct5) in
use structs_of_lists2_env with
 creates5_m:proc( -> struct5) ; creates5_n:proc( -> struct5) ; creates5_o:proc( -> struct5) ;
 creates5_p:proc( -> struct5) ; creates5_q:proc( -> struct5) ; creates5_r:proc( -> struct5) ;
 creates5_s:proc( -> struct5) ; creates5_t:proc( -> struct5) ; creates5_u:proc( -> struct5) ;
creates5_v:proc( -> struct5) in
 add5(add5(add5(add5(add5(add5(add5(add5(add5(add5(add5(
   add5(add5(add5(add5(add5(add5(add5(add5(add5(add5(add5(empty5(),
                                                          creates5_a()),
                                                        creates5_b()),
                                                      creates5_c()),
                                                    creates5_d()),
                                                  creates5_e()),
                                                creates5_f()),
                                              creates5_g()),
                                            creates5_h()),
                                          creates5_i()),
                                        creates5_j()),
                                      creates5_k()),
                                    creates5_l()),
                                  creates5_m()),
                                creates5_n()),
                              creates5_o()),
                            creates5_p()),
                          creates5_q()),
                        creates5_r()),
                      creates5_s()),
                    creates5_t()),
                  creates5_u()),
                creates5_v())

!PRINT_LIST5 PRINTS THE RULE BASE
let print_list5 = proc(l:list5)
while ~is_empty5(l) do
begin
 print_struct5(head5(l))
 l := tail5(l)
end

!CREATE_STRUCT2 CREATES THE INITIAL FACT BASE
let create_struct2 = proc( -> struct2)
 creates2("yes","","",empty1())

!ENGINE
in list2_env let treat_l5_s2 = proc(l:list5 ; s:struct2)
use list_env with is_empty12:proc(list12 -> bool) in
```

```
use list2_env with create12_with_l5s2:proc(list5,struct2 -> list12) ;
 get_log_1415:proc(struct12) ; firsts12_modified12:proc(list12 -> struct12) in
use sub_structs2_env with treat_s2_with_s12:proc(struct12,struct2 -> struct2) in
begin
 let finish := false
 while ~finish do
 begin
  let l1 := create12_with_l5s2(l,s)          !DETECTION
  if is_empty12(l1) then finish := true
  else
  begin
   let s1 := firsts12_modified12(l1)         !CHOICE
   get_log_1415(s1)
   s := treat_s2_with_s12(s1,s)              !DEDUCTION
  end
 end
end
```

Illustration of the operation creates5_a

Creates5_a creates the rule called move_adjacent2 and it uses an operation on struct5 (creates5), which in turn uses operations on list3 (create_list3a()), list4 (create_list4a()), variant7 (createv7_adjacent()), and list5 (empty5()).

```
let creates5_a = proc( -> struct5)
 creates5("move_adjacent2",create_list3a(),create_list4a(),createv7_adjacent(),
          "action_move_adjacent2",empty5())

!CREATE_LIST3A AND OTHER OPERATIONS IT USES
let create_list3a = proc( -> list3) ; add3(add3(empty3(),creates3cb()),creates3mm())

let add3 = proc(l:list3 ; a:struct3 -> list3) ; createv3a(creates3(sind3a(a),sind3b(a),l))

let empty3 = proc( -> list3) ; createv3b()

let creates3cb = proc( -> struct3) ; creates3("color","black",empty3())

let creates3mm = proc( -> struct3) ; creates3("move","machine",empty3())

!CREATE_LIST4A AND OTHER OPERATIONS IT USES
let create_list4a = proc( -> list4) ; add4(add4(empty4(),creates4_je()),creates4_ib())

let add4 = proc(l:list4 ; a:struct4 -> list4) ; createv4a(creates4(sind4a(a),sind4b(a),l))

let empty4 = proc( -> list4) ; createv4b()

let creates4_je = proc( -> struct4) ; creates4(createv2_j(),"empty",empty4())

let creates4_ib = proc( -> struct4) ;creates4(createv2_i(),"black",empty4())

let createv2_j = proc( -> variant2) ; createv2a("j")

let createv2_i = proc( -> variant2) ; createv2a("i")
```

```
!CREATEV7_ADJACENT AND OTHER OPERATIONS IT USES
let createv7_adjacent = proc( -> variant7) ; createv7a("adjacent")

!ACTION CORRESPONDING TO THE NAME OF THE ACTION GIVE IN CREATES5_A AND OTHER OPERATIONS IT USES
let action2_move_adjacent2 = proc(s:struct2 ; i,j:int -> struct2)
begin
 use global_env with move:list15 in move := add15_move(s,i,j,move)
 modify6_struct2(s,i,j,"empty","black")
end
let modify6_struct2 = proc(s:struct2 ; i,j:int ; a,b:string -> struct2)
begin
 s := sass2b(s,"user")
 s := sass2d(s,modify_list1(sind2d(s),i,j,a,b))
 s
end
let modify_list1 = proc(l:list1 ; p1,p2:int ; c1,c2:string -> list1)
begin
 l := modify_elem1(l,p1,c1)
 l := modify_elem1(l,p2,c2)
 l
end
rec let modify_elem1 = proc(l:list1 ; p:int ; c:string -> list1)
begin
 let AA := head1(l)
 if sind1a(AA) = p then add1(tail1(l),modify_struct1(AA,c))
 else add1(modify_elem1(tail1(l),p,c),AA)
end
let modify_struct1 = proc(s:struct1 ; c:string -> struct1) ; sass1b(s,c)

let head1 = proc(l:list1 -> struct1)
 creates1(sind1a(vind1a(l)),sind1b(vind1a(l)),createv1b())

let tail1 = proc(l:list1 -> list1) ; sind1c(vind1a(l))

let add1 = proc(l:list1 ; a:struct1 -> list1)
 createv1a(creates1(sind1a(a),sind1b(a),l))

let add15_move = proc( s:struct2 ; pos1,pos2:int ; move:list15 -> list15)
 append15(creates15(sind2b(s),pos1,pos2,empty15()),move)

rec let append15 = proc(s:struct15 ; move:list15 -> list15)
 if is_empty15(move) then add15(move,s)
 else add15(append15(s,tail15(move)),head15(move))

let is_empty15 = proc(l:list15 -> bool) ; vis15b(l)

let head15 = proc(l:list15 -> struct15)
 creates15(sind15a(vind15a(l)),sind15b(vind15a(l)),sind15c(vind15a(l)),createv15b())

let tail15 = proc(l:list15 -> list15) ; sind15d(vind15a(l))

let add15 = proc(l:list15 ; a:struct15 -> list15)
 createv15a(creates15(sind15a(a),sind15b(a),sind15c(a),l))
```

Appendix F

Previously Published Paper

On the Application of Software Engineering Techniques in Artificial Intelligence Research

Amaia Bernaras and Tim Smithers

Abstract It has become traditional for researchers in Artificial Intelligence o believe that Software Engineering has little or nothing to offer them in their work. They argue that the kind of program building they engage in is very different from that of the application program builder. We accept that there is a fundamental difference between program building in AI research and the kind of program building software engineering techniques have been primarily developed for. However, we also believe that at least some of the software engineering concepts and techniques currently being developed might be used to improve the research we do in AI. In particular, we suggest how abstraction constructs that are embedded in programming languages to help going from functional specifications to correct programs might help to go from experimental AI research programs to symbol level descriptions of them.

Presented in the Third International Workshop on Software Engineering and its Applications—Toulouse 1990 (available in pages 561–569 of the proceedings).

F.1 Introduction

It has become traditional for researchers in Artificial Intelligence (AI) to believe that Software Engineering (SE) has little or nothing to offer them in their work. They argue that the kind of program building they engage in, often called *fast prototyping*, is very different from that of the application program builder, and that it is only for these latter kinds of people that Software Engineering research is carried out. We accept that there is a fundamental difference between program building in AI research and the kind of program building SE techniques have been primarily developed for. However, we also believe that at least some of the SE concepts and techniques currently being developed might be used to improve the research we do in AI.

In this paper we first set out what we will mean by AI research — it doesn't include all the kinds of things people do in AI. We then describe the role of program building in AI research and contrast this with the life cycle of application program building. Finally, we suggest how certain concepts and techniques developed by programming language designers working in SE research might be used to improve AI research.

Our intention in submitting this paper to a Software Engineering conference is to invite comment on our view of how certain SE concepts and techniques might be used in AI research. We also see it as a small step towards bringing work in AI and SE research closer together: something we feel would benefit both parties.

F.2 The Nature of Artificial Intelligence Research

The proper nature and foundations of AI continues to be the subject of much debate, and rightly so for AI is still a relatively young discipline. For example, [Partridge & Wilks 90] presents a collection of papers from a workshop on the foundations of AI held in Las Cruces, New Mexico in 1986, together with some

earlier material to be found in the AI literature. For another example, see [Graubard 88], which presents a collection of invited papers first published in a *Daedalus* volume by the American Academy of Arts and Sciences.

Although these debates cover all kinds of AI, from symbolic functionalism to connectionism[1], in this paper we limit our use of the term AI research to the symbolic functionalist kind, or 'symbol processing-based AI' to use its more colloquial name. This has been the mainstream of AI research since its earliest days, and still represents the majority of AI research being carried out.

In order to develop the argument for how and why certain concepts and techniques developed by programming language designers in SE research can help improve research in AI, we first need to present AI as a science and then explain how symbolic functionalist research is carried out, or at least how it should be carried out. We start with AI as a science.

F.2.1 Artificial Intelligence as a Science

For us (and for many of our colleagues in AI) AI is a scientific discipline, not an engineering subject: building 'clever' programs *is not* the aim of AI research. Building programs is, however, the means by which much research is done. AI is the science which seeks to develop a theoretical understanding of intelligent behaviour by creating it in the artificial. This description embodies three different assumptions which we take to be true, and which we need to make explicit here. First, it assumes that science is concerned with developing theoretical understanding. Second, it assumes that AI is a science. Third, it assumes that intelligent behaviour is a valid subject of study.

The overall aim of a science is thus to develop theories about the phenomena

---

[1]A third '-ism' exists in AI research now, called robotic functionalism, this has yet to figure much in the mainstream of debate, but see [Harnad 90] for an attempt to suggest why it should.

under study. There are many ways to characterise a scientific theory. The one we will work with here is:

- A theory is some statement of a truth or a law about some aspect of the phenomenon under investigation which is expressible in a way independent of and without reference to particular examples or instances of the phenomenon.

There is another important dimension to doing science which we have not mentioned yet, that is, its empirical and experimental nature[2]. In all sciences the majority of effort goes into designing, executing, and evaluating experimental and/or empirical investigations. This often involves scientists in doing a lot of engineering. In AI research, this means building programs.

Although experimental and empirical investigation is important (indeed fundamental) to doing science, the process doesn't finish with just collecting and categorising data. It is necessary to use the understanding obtained to develop coherent explanations and theories which have predictive power. That is, to extract general laws or principles, or to offer evidence for some hypothesis that is being tested.

If we consider AI as a science, then we need to be able to develop a theoretical understanding of the intelligent behaviour being investigated. How this is done in symbolic functionalism is described in the next section.

F.2.2 Symbolic Functionalism

Symbolic Functionalism in AI is based upon Newell and Simon's *Physical Symbol System Hypothesis* (PSSH). This was first presented by them in the tenth Turing award lecture, delivered to the annual conference of the Association for Computing Machinery (ACM) in 1975, and subsequently published in the

[2]It is philosophy that is the discipline which attempts to establish truths about the world by purely abstract means.

Communications of the ACM, [Newell & Simon 76]. It forms a general scientific hypothesis, a law of qualitative structure for symbol systems, and it states that:

> A physical symbol system has the necessary and sufficient means for general intelligent action.

Here (according to Newell and Simon), the term *necessary* means that any system that exhibits general intelligence will prove upon analysis to be a physical symbol system. *Sufficient* means that any physical symbol system of sufficient size can be organized further to exhibit general intelligence. The expression *general intelligent action* denotes the same scope of intelligence as seen in human action. That is, to have an appropriate behavior in any real situation. The appropriateness of the behaviour is measured in terms of the ends of the system and its adaptiveness to the demands of the environment, within some limits of speed and complexity.

Newell and Simon consider this Hypothesis as an empirical generalization, not a theorem. They describe AI as addressing itself to the sufficiency of Physical Symbol systems for engendering intelligent behaviour, attempting to construct and test specific systems that have such a capability. When addressing the problem of building such systems, the first thing we need to know is what is the class of systems the hypothesis defines. Newell and Simon describe the hypothesis as specifying a general class of systems within which those capable of intelligent action can be found. Intelligent action is a form of behaviour that can be recognized by its effects, but it is not so easy to produce such that any system will exhibit it willy-nilly. In this way Newell and Simon argue that intelligent behaviour is a substantial phenomenon which can be engendered by a suitably constructed physical symbol system, in other words, a computer program of the appropriate kind. The question is what kind of computer program?

If we are to take Newell and Simon's PSSH seriously, that is, if we are to investigate the validity of this hypothesis, and thus its possible status as a theoretical statement about intelligent behaviour, AI must be practiced as a science not simply as an engineering activity. So, if we build programs as part

of our empirical investigation, they can't be allowed to stand as black boxes demonstrating some kind of intelligent behaviour, such as chess playing, for example. We need to be able to explain what these programs do, how they do it, and why they have to do it to play chess, etc. They should be understandable systems, understandable in terms of the computational concepts they employ and depend upon to realise the intelligent behaviour they do. For this to be the case AI research programs need to be based upon well founded computational concepts. An AI program whose behaviour can only be explained in terms of the many lines of Lisp or Prolog code that constitutes it is of no scientific use whatever. Worse, it can seriously distract much of the efforts of those researchers who try to extend or change it.

F.2.3 Software Engineering Research

Understanding the computational concepts required to engender particular kinds of behaviour in computer programs is one of the concerns of Software Engineering research. We take SE research to be the discipline concerned with the formalization of software production. The fundamental aim is to develop formal concepts and general principles, mechanisms, and techniques to understand computational systems and the relations among system components, as well as the mathematical modelling techniques associated with these systems.

As AI researchers we should therefore be looking to make good use of those developments in SE research that can help us to better understand the programs we build to experimentally investigate Newell and Simon's PSSH.

F.3 The SE Problem in AI Research

As we have said, AI research should be trying to experimentally investigate the validity of the PSSH. This means attempting to build examples of Physical Symbol Systems (PSSs) and testing them with respect to the intelligent behaviour they are intended to engender. To construct PSSs and test them we build

computer programs; it is therefore important that we are able to establish and understand what PSS a particular program actually realises.

To explain and distinguish the steps taken in this process, it is first necessary to mention the three levels at which AI research operates. These are:

- The *Knowledge Level.* This level treats knowledge as a competence notion, a potential to act. It specifies what kind of knower a system needs to be to behave intelligently, and what knowledge is required to be such a knower[3].

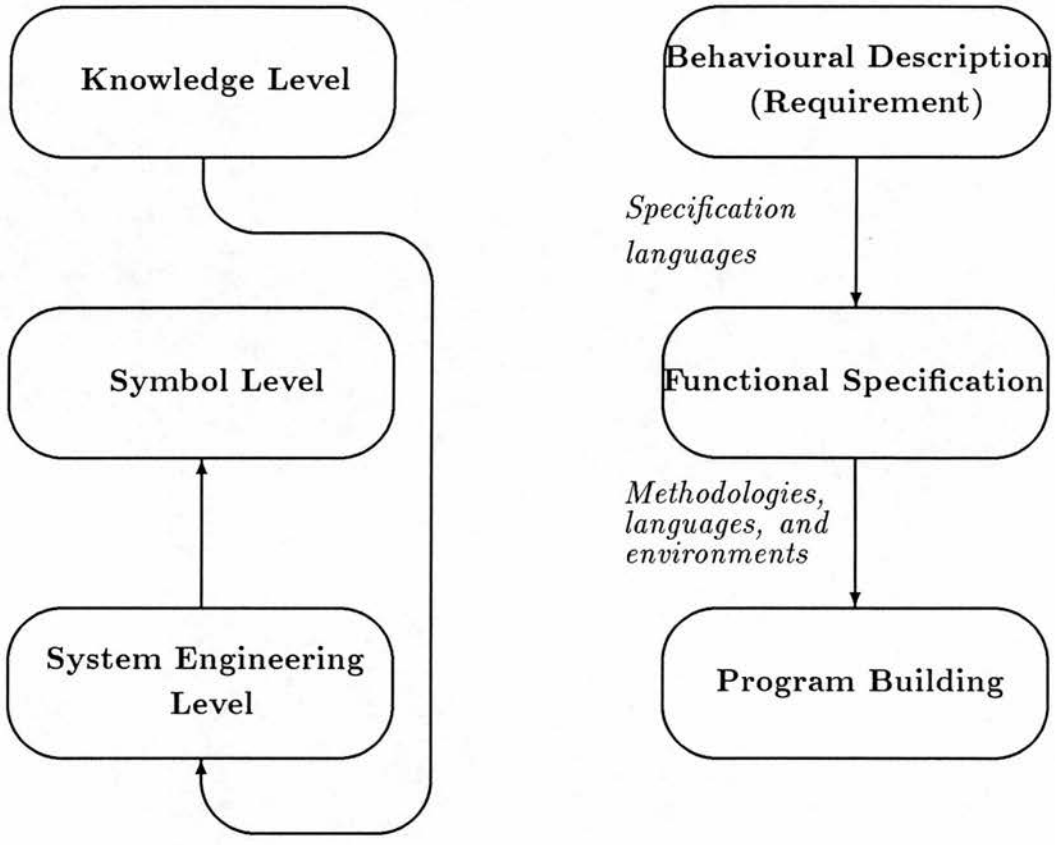
- The *Symbol Level.* This level is concerned with the description of the symbol system required to engender the behaviour specified at the knowledge level. It is at the Symbol Level that we should explain how and why a particular program, or kind of program, realised the intelligent behaviour it does.

- The *System Engineering level.* This level is concerned with how the symbol system is to be physically realised so that its actual behaviour can be experimentaly tested and assessed with respect to the knowledge level description. It is at the System Engineering Level that we are operating at when we build our research programs.

There are many problems to be overcome in this process. The first one is that, deciding at the knowledge level what behaviour to investigate does not tell us what PSS is required to engender it. We don't even know if there is one, this is what we are trying to discover. As a consequence, we do not have a well formed and complete specification for the programs we build, though we usually have some idea of what it could be like and how it might be built. Building AI programs and testing them is thus the process by which we try to discover if there is a PSS which can engender the intelligent behaviour being investigated. If there is, we then need to specify it in formal terms at the Symbol Level. This

---

[3]The level was first presented by Newell in his 1980 presidential address to the American Association for Artificial Intelligence (AAAI), see [Newell 81] for a published version.



1a. AI Research

1b. Application Building

Figure F–1: AI research life cycle compared to the application building life cycle.

is the level at which PSSs are to be described. It is therefore at this level that we should be collecting evidence for and/or against the PSSH.

For this methodology to work we must be able to understand the programs we write in a way that enables us to subsequently abstract from and specify the PSSs they actually realise. In other words, experimentally investigating the validity of the PSSH doesn't just mean building programs which do interesting things, we must also say what PSSs are being realised by our programs. Their behaviour, after all, may not depend upon symbol processing. In which case doubt may be cast upon the PSSH as a candidate theory of intelligent behaviour.

If we consider this process as a three step process (see figure 1a), each of which

may contain substeps. The first step consists of deciding, at the Knowledge Level, what intelligent behaviour we want to investigate. The second step is to try to build a program to realise this behaviour. This step is carried out at the System Engineering Level. This program is the *input* to the third step in the process in which we try to discover and formalise the PSS which the program realises, and which thus engenders the Knowledge Level behaviour being investigated. The problem is to find a way of correctly abstracting from the program to the symbol system description it implements.

In contrast, when building application programs, the aim and thus the methodology followed is different (see figure 1b). In this case the process can again be described in terms of three steps. Starting from a description of the problem to be solved, the program builder constructs a description of the required behaviour. In the second step, sometimes done using a specification language, the requirement description is translated into a set of formal specifications that define the functionality of the program required to deliver the desired behaviour. Finally, in the third step, the functional specification is used as the input to the program building process that will realise the solution to the application problem. For this last step different methodologies, languages, and environments may be used.

Research in programming language design is primarily motivated by a desire to make the process of going from a functional specification of a program to an actual implementation easier and less arbitrary, i.e. more formal. It has been seeking to do this by providing more powerful and more coherently combined computational constructs, such as strong data typing, polymorphism, and persistence, for example.

Modern programming languages which embody such concepts enable program builders to work at more abstract levels of programming, thus freeing them from finer grained details that are not of specific relevance to the functionality of the particular program being built. If such techniques can help in going from a functional specification to a program, then perhaps they can also help in abstracting from a program to a symbol level description of the PSS it realises.

In other words, if going from a functional specification to a correct program is equivalent to but in the reverse direction of going from a program to a symbol system description, then the computational concepts and techniques designed to help the former might also help the latter.

F.4 A Way of Engineering Better Programs in AI Research

There is concern in the SE community about the need to construct programs that are correct with respect to a given specification. Correctness and specification accomplishment are two of the most important motivations behind research in SE.

These two requirements are specially difficult to meet when building large complex systems. Research in the area has shown that it is necessary to control the complexity of a system to allow the developer to concentrate on the specified functionality rather than on the details of construction imposed by the complexity of the system, but that are irrelevant to the specification. A well known method of controlling complexity is abstraction, where the details of a particular level can be ignored when viewing the system from a higher level. In this sense, part of the research effort, [Morrison *et al* 88], has concentrated on the development of programming paradigms and concepts that allow degrees of abstraction over different complexity introducing aspects, time and data, for example.

Two abstraction mechanisms that are important when building complex systems are persistence and strong typing. Persistence abstracts over the length of time that data exists and is usable [Atkinson *et al* 83]. The persistent programming paradigm aims to manipulate long term data in the same manner as short term data. It provides the program builder with a uniform model of data over time, yielding an integration of the programming language and its environment. The second abstraction mechanism is that of abstract data type. The concept of data type is usually realised by a type system that

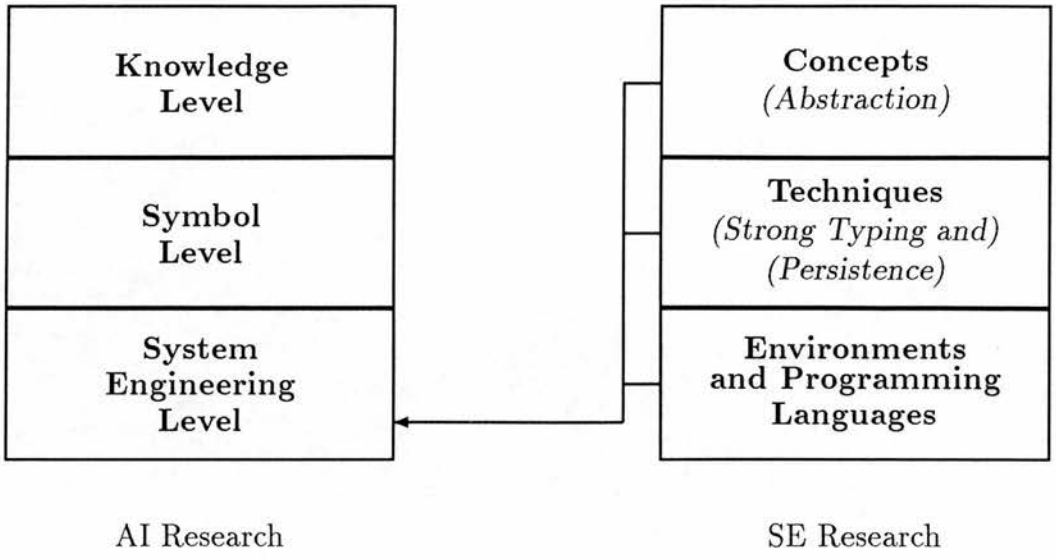


Figure F–2: Interaction between AI and SE research

may have its semantic roots in the mathematical notion of sets. For example, [Cardelli & Wegner 85] explains how the universe of all computable objects V, decomposes naturally into subsets with uniform behaviour. Sets of objects with uniform behaviour can be named and referred to as types. It is important to note that not all subsets of the universe V are legal types, they must obey some technical properties. The usefulness of a type system lies not only in the set of types that can be represented, but also in the kinds of relationships among types that can be expressed, and thus the abstractions which can be built and used. The ability to express relations among types involves some ability to perform computations on types to determine whether they satisfy the desired relationship, i.e. type equivalence, type inclusion, etc.

Figure F–2 shows how these concepts and paradigms are realised by computational constructs and techniques that are usually implemented by embedding them in a programming language. The arrow connecting the three levels of research in SE with the System Engineering Level in AI research indicates the idea that these concepts may help in abstracting from a experimental program, built using a programming language which embodies such computational constructs, to a Symbol Level description of the PSS the program perhaps realises.

F.5 From Programs to Symbol Systems

A programming language embodying powerful abstraction mechanisms can be used in several ways when building programs. One way would be to begin building our experimental programs by making as much use as possible of its abstracting constructs available in the language.

Another way which, in our opinion, is more appropiate for our purposes would be to employ the abstraction constructs in a transformation process which takes an initial (working) program into a more abstractly expressed form which preserves its behaviour, and thus brings it nearer to a suitable Symbol Level description of it. In other words, it is only after a program that behaves in some interesting way has been built (and possibly after the several attempts typically needed to build such a program), using or not the abstraction constructs available, do we attempt to transform it into a more abstractly expressed program using the available abstraction constructs. We envisage this being done in a series of stages. In each stage the program is transformed into a more abstract expression and tested for behaviour preservation by executing it, which we can do because, although it is a more abstract description of the initial program, it is still an executable program. The process finishes when we can't make the program more abstract as well as testable by executing it.

It is perhaps by such a process of testable abstraction that we might be able to more easily go from some experimental program, built to engender the intelligent behaviour we are interested in, to a Symbol Level description of it. It is only by doing this that we can scientifically investigate the validity of Newell and Simon's PSSH.

F.6 Conclusions

In this paper we have suggested how certain concepts and techniques developed by programming language designers in Software Engineering research might be used to help AI research. We started by describing the nature of AI research, in its symbolic functionalist form, and described the three stage process by which a symbol system description of programs built to engender a particular kind of intelligent behaviour is derived. We then presented a proposal for how the process of going from a program to its symbol level description might be aided using abstraction constructs embodied in programming languages intended to aid the process of going from a functional specification to a correct program.

Acknowledgements

The work presented here has been carried out within the AI in Design research group in the Department of Artificial Intelligence at Edinburgh, and we gratefully acknowledge the the other members of this team who have contributed to discussions of the ideas expressed. We would also like to thank Professor Ron Morrison and other members of the Persistent Programming group at St Andrews University for first bring to our attention the possible relevance of modern Software Engineering research, and for subsequent support in the form of the PS-Algol and Napier88 programming languages. Lastly, but not least, we acknowledge the financial support of the Spanish Government for Amaia Bernaras.

Bibliography

[Andersson 88] Russell L. Andersson. *A Robot Ping-Pong Player: Experiment in Real-Time Intelligent Control.* The MIT Press, Cambridge, Massachusetts, 1988.

[Atkinson & Morrison 88] Malcolm P. Atkinson and Ron Morrison. *Types, Bindings and Parameters in a Persistent Environment. In Data Types and Persistence*, chapter 1, pages 3–20. Springer-Verlag, Berlin, 1988.

[Atkinson *et al* 83] Malcolm P. Atkinson, Peter J. Baley, W. P. Cockshott, and Ron Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, November 1983.

[Atkinson *et al* 84] Malcom P. Atkinson, P. Baley, W. P. Cockshott, K. J Chisholm, and Ron Morrison. Progress with persistent programming. In P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, editors, *Databases—Role and Structure*, pages 245–310. Cambridge University press, 1984.

[Balkany *et al* 91] A. Balkany, W. P. Birmingham, and I. D. Tommelein. A knowledge-level analysis of several design tools. In John Gero, editor, *Artificial Intelligence in Design'91*, pages 921–940. Butterworth-Heinemann, 1991.

[Barbedette 90] Giles Barbedette. LISPO2: A persistent object-oriented lisp. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology–EDBT'90*, pages 332–347. Springer-Verlag, March 1990.

[Barr & Feigenbaum 81] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume 1. Addison-Wesley, 1981.

[Bell 83] R. C. Bell. *The Boardgame book.* Marshall Cavendish books Ltd., second edition, 1983. First edition 1979.

[Berliner & Ebeling 89] H. J. Berliner and C. Ebeling. Pattern knowledge and search: The SUPREM architecture. *Artificial Intelligence*, 38(2), 1989.

[Bernaras & Smithers 90] Amaia Bernaras and Tim Smithers. On the application of software engineering techniques in artificial intelligence research. In *Proceedings of the Third International Workshop on Software Engineering and its Applications–Toulouse 90*, pages 561–569. EC2, December 1990. Also available as DAI Research Paper No. 503, Department of Artificial Intelligence, University of Edinburgh, 1990.

[Brachman & Levesque 86] Ronald Brachman and Hector Levesque. The knowledge level of KBMS. In M. L. Brodie and L. Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Artificial Intelligence and Database Technologies*, pages 9–12. Springer-Verlag—Topics in Information Systems, 1986.

[Bramer & Cendrowska 84] Max A. Bramer and Jadzia Cendrowska. Inside an expert system: A rational reconstruction of the MYCIN consultation system. In M. Eisenstadt and T. O'Shea, editors, *Artificial Intelligence: Tools and Techniques*, pages 453–497. New York: Harper and Row, 1984.

[Buchanan & Shortliffe 84] Bruce G. Buchanan and E. H. Shortliffe. *Rule-Based Expert Systems: the MYCIN Experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, MA, 1984.

[Buchanan 88] Bruce G. Buchanan. Artificial intelligence as an experimental science. In James H. Fetzer, editor, *Aspects of Artificial Intelligence*, pages 209–250. Kluwer Academic Publishers, 1988.

[Bundy 90] Alan Bundy. What kind of field is AI? In Derek Partridge and Yorick Wilks, editors, *The Foundations of Artificial Intelligence*, pages 215–222. Cambridge University Press, 1990.

[Bundy *et al* 90] Alan Bundy, Fausto Giunchiglia, and Toby Walsh. Building abstractions. DAI Research Paper 506, University of Edinburgh. Department of Artificial Intelligence, 1990.

[Burstall & Goguen 82] R. Burstall and J. Goguen. Algebras, theories and freeness: An introduction for computer scientists. CSR-101-82, Dept. of Computer Science. University of Edinburgh, 1982.

[Campbell 90] J. A. Campbell. Three novelties of AI: theories, programs, and rational reconstructions. In Derek Partridge and Yorick Wilks, editors, *The Foundations of Artificial Intelligence*, pages 237–246. Cambridge University Press, 1990.

[Cardelli & Wegner 85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[Cardelli 84] L. Cardelli. Amber. Technical report, AT&T Bell Laboratories, Murray Hill, N.Y., 1984.

[Carrick *et al* 87] Ray Carrick, Jack Cole, and Ron Morrison. An introduction to ps-Algol programming. Persistent Programming Research 31, Department of Computational Science, University of St. Andrews, Scotland, October 1987.

[Chalmers 82] A.F. Chalmers. *What is this thing called Science?* Open University Press, second edition, 1982.

[Charniak & McDermott 85] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Computer Science. Addison-Wesley, 1985.

[Chikofsky & Cross 90] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[Choi & Scacchi 90] S.C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, January 1990.

[Clocksin & Mellish 81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[Cockshott *et al* 83] W. P. Cockshott, Malcolm P. Atkinson, Ken Chrisholm, Peter J. Baley, and Ron Morrison. The persistent object management system. *Software Practice and Experience*, 14(1):49–71, 1983. Also as Technical Report PPR-1-83.

[Cohen 91] P. R. Cohen. A survey of the eighth national conference on artificial intelligence: Pulling together or pulling apart? *AI Magazine*, 12(1):17–41, 1991.

[Colomb 88] Robert M. Colomb. Issues in the implementation of a persistent prolog. TR-FB-88-03, CSIRO Division of Information Technology, Australia, November 1988.

[Cummins 89] Robert Cummins. *Meaning and Mental Representations.* MIT Press (A Bradford book), 1989.

[Dahl & Nygaard 66] O. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[Danforth & Tomlinson 88] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM computing Surveys*, 20(1):29–72, March 1988.

[Darlington 86] J. Darlington. An experimental program transformation and synthesis system. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, pages 99–122. Morgan Kaufmann, 1986.

[Dearle *et al* 90] Alan Dearle, Richard Connor, Fred Brown, and Ron Morrison. Napier88–a database programming language? In *Type Systems and Database Programming Languages*, pages 10–26. Research Report CS/90/3. University of St. Andrews, 1990.

[deKleer & Williams 87] J. de Kleer and B.C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[Demers & Donahue 79] A. Demers and J. Donahue. Revised report on Russell. Technical report TR-79-389, Dept. of Computer Science, Cornell Univ. Ithaca, N.Y, 1979.

[Fikes 90] Richard Fikes. AI and software engineering—managing exploratory programming. In *Proceedings of AAAI-90. National Conference on Artificial Intelligence. In Invited Talks and Panels. Special Panel on AI and SE*, pages 1126–1127. American Association for Artificial Intelligence, July 1990.

[Floyd 84] Christiane Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer-Verlag. Berlin, 1984.

[Ford 87] L. Ford. Artificial intelligence and software engineering: A tutorial introduction to their relationship. *Artificial Intelligence Review*, 1(4):255–273, 1987.

[Frühwirth 88] T. M. Frühwirth. Type inference by program transformation and partial evaluation. In *Proceedings of Meta'88*, pages 199–212, 1988.

[Futatsugi *et al* 85] K. Futatsugi, J. Goguen, J. P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, New York, 1985.

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[Graubard 88] Stephen R. Graubard, editor. *The Artificial Intelligence debate: False starts, real Foundations.* MIT Press, 1988.

[Harandi & Ning 90] Mehdi T. Harandi and Jim Q. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–80, January 1990.

[Harnad 89] Stevan Harnad. Minds, machines and searle. *JETAI*, 1:5–25, 1989.

[Harnad 90] Stevan Harnad. The symbol grounding problem. *Physica D*, pages 335–346, 1990. 42(1–3).

[Hayes 75] Patrick J. Hayes. Nine deadly sins. *AISB European Newsletter*, pages 15–17, July 1975.

[Hughes 88] John G. Hughes. *Database Technology: A Software Engineering Approach.* Series in Computer Science. Prentice Hall International, 1988.

[Ichbiah *et al* 79] J. H. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Bruckner, O. Roubine, and B. A. Wich-Mann. Rationale of the design of the programming language Ada. *ACM Sigplan Notices*, 14(6), 1979.

[Jackson 85] Philip C. Jackson. *Introduction to Artificial Intelligence.* Dover publications, Inc., New York, second edition, 1985.

[Johnson & Soloway 86] W. Johnson and E. Soloway. Proust: Knowledge-based program understanding. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, pages 443–451. Morgan Kaufmann, 1986.

[Johnson 86] W. Johnson. *Intention-Based Diagnosis of Novice Programming Errors.* Morgan Kaufmann, 1986.

[Jones 90] Karen Sparck Jones. What sort of thing is an AI experiment? In Derek Partridge and Yorick Wilks, editors, *The Foundations of Artificial Intelligence*, pages 274–281. Cambridge University Press, 1990.

[Kernighan & Plauger 74] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style.* New York NY: McGraw-Hill, 1974.

[Kirby & Dearle 90] G. N. C. Kirby and Alan Dearle. An adaptive graphical browser for Napier88. CS/90/16, Department of Computational Science, University of St. Andrews, Scotland, 1990.

[Knapman 78] J. Knapman. A critical review of Winston's learning structural descriptions from examples. *AISB Quarterly*, 31:18–23, 1978.

[Knowles & Bell 84] J. S. Knowles and D. M. R. Bell. The CODASYL model. In P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, editors, *Databases—Role and Structure*, pages 19–56. Cambridge University Press, 1984.

[Kreutzer & Mckenzie 91] Wolfgang Kreutzer and Bruce Mckenzie. *Programming for Artificial Intelligence. Methods, Tools and Applications.* Addison-Wesley, 1991.

[Laird *et al* 86] John E. Laird, Paul S. Rosenbloom, and Allen Newell. *Universal Subgoaling and Chunking. The Automatic Generation and Learning of Goal Hierarchies.* Kluwer Academic Publishers, 1986.

[Laird *et al* 87] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987. Also available as Technical Report 2. University of Michigan. Cognitive Science and Machine Intelligence Laboratory.

[Lakatos 74] I. Lakatos. Falsification and the methodology of scientific research programmes. In I. Lakatos and A. Musgrave, editors, *Criticism and the Growth of Knowledge*, pages 91–196. Cambridge University Press, 1974.

[Ledgard 75] H. F. Ledgard. *Programming Provers.* Rochelle Park NJ: Hayden, 1975.

[Lenat 76] Douglas B. Lenat. An artificial intelligence approach to discovery in mathematics as heuristic search. Stanford AI laboratory Memo 286, Stanford University, Stanford CA: Department of Computer Science, 1976.

[Logan *et al* 91] B. Logan, K. Millington, and T. Smithers. Being economical with the truth: assumption-based context management in the Edinburgh Designer System. In John Gero, editor, *Artificial Intelligence in Design'91*, pages 423–446. Butterworth–Heinemann Ltd., 1991.

[Lopez *et al* 90] B. Lopez, P. Meseguer, and E. Plaza. Knowledge based systems validation: A state of the art. *AI Communications*, 3(2):58–72, June 1990.

[Luger & Stubblefield 89] George F. Luger and William A. Stubblefield. *Artificial Intelligence and the Design of Expert Systems.* The Benjamin/Cummings Publishing Company Inc., 1989.

[Lukey 80] F. J. Lukey. Understanding and debugging programs. *Intelligent Journal of Man-Machine Studies*, pages 189–202, 1980.

[MacQueen 86] D. MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 277–286. ACM, New York, 1986.

[Marr 82] David Marr. *Vision.* W. H. Freeman and Co., 1982.

[Martin-Löf 82] P. Martin-Löf. Constructive logic and computer programming. In *Proceedings of the 6th International Congress for Logic, Methodology, and Philosophy of Science.* North-Holland Amsterdam, 1982.

[Matthews 88] D. C. J. Matthews. An overview of the Poly programming language. In M. P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 43–50. Springer-Verlag, 1988.

[McCarthy & Hayes 69] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[McCarthy 62] J. McCarthy. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass, 1962.

[McCarthy 83] J. McCarthy. President's quarterly message: AI needs more emphasis on basic research. *AI Magazine*, 4(4), 1983.

[McCarthy 85] J. McCarthy. Epistemological problems of artificial intelligence. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 23–30. Morgan Kaufmann Publishers, 1985.

[McDermott 87] Drew McDermott. A critique of pure reason. *Computational Intelligence*, 3(3):151–160, 1987.

[Milner 83] Robin Milner. A proposal for standard ML. *Polymorphism*, 1(3), December 1983.

[Mitchell & Plotkin 85] J. Mitchell and G. Plotkin. Abstract types have existential type. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 37–51. ACM, New York, 1985.

[Moffat & Gray 88] David S. Moffat and Peter M. D. Gray. Perlog: A prolog with persistence and modules. *The Computer Journal*, 31(2):110–115, 1988.

[Morgan 90] Carroll Morgan. *Programming from Specifications.* Computer Science. Prentice Hall, 1990.

[Morrison *et al* 87] Ron Morrison, Malcolm P. Atkinson, and Alan Dearle. Flexible incremental bindings in a persistent object store. Persistent Programming Research Report 38, University of St. Andrews, Scotland, Department of Computational Science, June 1987.

[Morrison *et al* 88] Ron Morrison, Alan Dearle, and C. Marlin. Adaptive data stores. Persistent Programming Research Report 66, University of St. Andrews, Scotland, Department of Computational Science, October 1988.

[Morrison *et al* 89] Ron Morrison, Fred Brown, Richard Connor, and Al Dearle. The Napier88 reference manual. Technical report, Universities of Glasgow and St. Andrews, Scotland, July 1989.

[Murray 85] W. R. Murray. Heuristic and formal methods in automatic program debugging. In *Proc. Ninth International Joint Conference on Artificial Intelligence*, pages 15–19. Morgan Kaufmann, Los Altos California, 1985.

[Neumann 92] Bernd Neumann, editor. *Proceedings of the 10th European Conference on Artificial Intelligence. ECAI–92.* ECCAI, John Wiley & Sons, August 1992.

[Newell & Simon 72] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Newell & Simon 76] Allen Newell and Herbert A. Simon. Computer science as an empirical inquiry: Symbols and search. *Communications of the ACM*, 19:113–126, March 1976. Also in Mind Design, John Haugeland (ed.), Cambridge Mass.: The MIT Press (A Bradford Book).

[Newell 80] Allen Newell. Physical symbol systems. *Cognitive Science*, 4:135–183, 1980.

[Newell 81] Allen Newell. The knowledge level. CMU-CS-81-131, Carnegie-Mellon University. Dept. of Computer Science, July 1981. Also published in Artificial Intelligence, 18(1), pages 87–127, 1982.

[Palsberg & Schwartzbach 91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA'91 Proceedings*, pages 146–161, 1991.

[Parsaye *et al* 89] Kamran Parsaye, Mark Chignell, Setrag Khoshafian, and Harry Wong. *Intelligent Databases. Object-Oriented, Deductive Hypermedia Technologies.* John Wiley and Sons, Chichester, 1989. ISBN 0-471-50346-0.

[Partridge & Wilks 90] Derek Partridge and Yorick Wilks, editors. *The Foundations of Artificial Intelligence.* Cambridge university press, 1990.

[Partridge 86] Derek Partridge. Engineering artificial intelligence software. *Artificial Intelligence Review*, 1:27–41, 1986.

[Pascoe 86] Geoffrey A. Pascoe. Elements of object-oriented programming. *Byte*, 11(8):139–144, August 1986.

[Popper 68] Karl Popper. *The Logic of Scientific Discovery.* Hutchinson, 1968.

[Preece 90] Alun D. Preece. Towards a methodology for evaluating expert systems. *Expert Systems*, 7(4):215–223, November 1990.

[Rauch-Hindin 88] Wendy B. Rauch-Hindin. *A Guide to Commercial Artificial Intelligence.* Prentice-Hall, 1988.

[Rich & Waters 86] Charles Rich and Richard C. Waters, editors. *Readings in Artificial Intelligence and Software Engineering.* Morgan Kaufmann, Los Altos, California, 1986.

[Rich & Waters 90] Charles Rich and Richard C. Waters. *The Programmer's Apprentice.* New York ACM Press Reading, Mass. Wokingham Addison-Wesley, 1990.

[Rich & Wills 90] Charles Rich and Lind M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, pages 82–89, January 1990.

[Rich 81] Charles Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of the IJCAI-81*, 1981.

[Rich 83] Elaine Rich. *Artificial Intelligence.* Computer Science. Mc Graw-Hill International Editions, 8th - printing 1988 edition, 1983. ISBN 0-07-052261-8.

[Richardson & Carey 89] J. Richardson and M. Carey. Persistence in the e language: Issues and implementation. *Software–Practice & Experience*, 19(12), 1989.

[Ritchie & Hanna 84] Graeme D. Ritchie and F. K. Hanna. AM: A case study in AI methodology. *Artificial Intelligence*, 23(3):249–269, 1984.

[Ritchie 91a] Graeme Ritchie. Learning from AM. DAI Research paper 522, Department of Artificial Intelligence, University of Edinburgh, 1991.

[Ritchie 91b] Graeme Ritchie. Programming in lisp. Lectures notes for the MSc course on Lisp. Department of Artificial Intelligence, University of Edinburgh, 1991.

[Roberts 89] Gary Allen Roberts. A rational reconstruction of Wilson's Animat and Holland's CS-1. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 317–321. Morgan Kaufmann Publishers Inc., 1989.

[Roberts 91] Gary Allen Roberts. *Classifier Systems for Situated Autonomous Learning.* Unpublished PhD thesis, University of Edinburgh. Department of Artificial Intelligence, 1991.

[Rosembloom *et al* 91] Paul S. Rosembloom, John E. Laird, Allen Newell, and Robert McCarl. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47:289–325, 1991.

[Ruth 86] G. Ruth. Intelligent program analysis. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, pages 431–442. Morgan Kaufmann, 1986.

[Schmidt 77] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–281, September 1977.

[Schank 90] Roger C. Schank. What is AI anyway? In Derek Partridge and Yorick Wilks, editors, *The foundations of Artificial Intelligence*, pages 3–13. Cambridge university press, 1990.

[Shipman 81] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[Simon 81] Herbert A. Simon. *The Sciences of the Artificial.* MIT Press, Cambridge MA, second edition, 1981. First published in 1969.

[Smith 85] Brian C. Smith. Prologue to reflection and semantics in a procedural language. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 31–39. Morgan Kaufmann Publishers, Inc., 1985.

[Smith *et al* 81] J. M. Smith, S. Fox, and T. Landers. *Reference Manual for ADAPLEX.* Computer Corporation of America. Cambridge, Massachusetts, 1981.

[Smithers 87] Tim Smithers. The Alvey large scale demonstrator project Design to Product. In Thomas Bernold, editor, *Artificial Intelligence in Manufacturing: Key to Integration?*, pages 251–261. North-Holland, 1987.

[Smithers 91] Tim Smithers. Lectures notes for the MSc course on knowledge representation and inference two. Department of Artificial Intelligence, University of Edinburgh, 1991.

[Smithers *et al* 89] Tim Smithers, Alistair Conkie, Jim Doheny, Brian Logan, and Karl Millington. Design as intelligent behaviour: An AI in design research programme. In John S. Gero, editor, *Design Theme of the Fourth International Conference on Applications of Artificial Intelligence in Engineering*, pages 293–334, Cambridge, England, July 1989. Computational Mechanics.

[Smithers *et al* 92] T. Smithers, M.X. Tang, N. Tomes, P. Buck, B. Clarke, G. Lloyd, K. Poulter, C. Floyd, and E. Hodgkin. Exploration and reflection: AI-based design research in the castlemaine project. *Knowledge-Based Systems Journal*, March 1992. Special Issue.

[Smolensky 88] P. Smolensky. On the proper treatment of connectionism. *The Behavioural and Brain Sciencies*, 11(1):1–23, 1988.

[Steels 90] Luc Steels. Components of expertise. *AI Magazine*, 11(2):28–49, Summer 1990. Also as a technical report of the VUB AI Lab, AI Memo 89-2.

[Ruth 86] G. Ruth. Intelligent program analysis. In Charles Rich and Richard C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, pages 431–442. Morgan Kaufmann, 1986.

[Schmidt 77] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–281, September 1977.

[Schank 90] Roger C. Schank. What is AI anyway? In Derek Partridge and Yorick Wilks, editors, *The foundations of Artificial Intelligence*, pages 3–13. Cambridge university press, 1990.

[Shipman 81] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[Simon 81] Herbert A. Simon. *The Sciences of the Artificial.* MIT Press, Cambridge MA, second edition, 1981. First published in 1969.

[Smith 85] Brian C. Smith. Prologue to reflection and semantics in a procedural language. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 31–39. Morgan Kaufmann Publishers, Inc., 1985.

[Smith *et al* 81] J. M. Smith, S. Fox, and T. Landers. *Reference Manual for ADAPLEX.* Computer Corporation of America. Cambridge, Massachusetts, 1981.

[Smithers 87] Tim Smithers. The Alvey large scale demonstrator project Design to Product. In Thomas Bernold, editor, *Artificial Intelligence in Manufacturing: Key to Integration?*, pages 251–261. North-Holland, 1987.

[Smithers 91] Tim Smithers. Lectures notes for the MSc course on knowledge representation and inference two. Department of Artificial Intelligence, University of Edinburgh, 1991.

[Smithers *et al* 89] Tim Smithers, Alistair Conkie, Jim Doheny, Brian Logan, and Karl Millington. Design as intelligent behaviour: An AI in design research programme. In John S. Gero, editor, *Design Theme of the Fourth International Conference on Applications of Artificial Intelligence in Engineering*, pages 293–334, Cambridge, England, July 1989. Computational Mechanics.

[Smithers *et al* 92] T. Smithers, M.X. Tang, N. Tomes, P. Buck, B. Clarke, G. Lloyd, K. Poulter, C. Floyd, and E. Hodgkin. Exploration and reflection: AI-based design research in the castlemaine project. *Knowledge-Based Systems Journal*, March 1992. Special Issue.

[Smolensky 88] P. Smolensky. On the proper treatment of connectionism. *The Behavioural and Brain Sciencies*, 11(1):1–23, 1988.

[Steels 90] Luc Steels. Components of expertise. *AI Magazine*, 11(2):28–49, Summer 1990. Also as a technical report of the VUB AI Lab, AI Memo 89-2.